Cocoa Event Handling Guide

Developer

Contents

Figures, Tables, and Listings

Introduction

A primary responsibility of an event-driven application is to handle user events—that is, events generated by devices such as mice, keyboards, trackpads, and tablets. For most Cocoa applications, the Application Kit assumes the largest share of this work. It ensures that events generated by the mouse, keyboard, and other devices are routed to the objects best suited to handle them. It also implements dozens of user-interface objects such as controls and text views to respond to events in expected ways—for example, by inserting typed text or sending an action message. But often an application, especially an application with custom `NSView`, `NSWindow`, or `NSApplication` objects, finds that it must handle some events itself.

Cocoa Event Handling Guide explains how to handle events of all types in a Cocoa application. It provides conceptual background for the task-based chapters by describing the Cocoa architecture for dispatching and handling events, and by giving an overview of `NSEvent` objects, which all event-handling code must deal with. Reading this document will give you a solid foundation for handling events in your Cocoa application. It is recommended that you read *Cocoa Fundamentals Guide* before reading this document.

Organization of This Document

This document includes the following chapters:

- "Event Architecture" (page 10) describes how events enter a Cocoa application, how they are dispatched to view objects, and how they are handled, sometimes after traveling up a chain of responder objects.
- "Event Objects and Types" (page 27) examines the Cocoa objects that represent events and surveys the types of events a Cocoa application can receive.
- "Event Handling Basics" (page 39) presents the fundamental tasks in event-handling code regardless of event type.
- "Handling Mouse Events" (page 51) describes how you can handle events arising from the user clicking or dragging the mouse.
- "Handling Key Events" (page 66) describes how you can handle events resulting from the user pressing keys on a keyboard.
- "Using Tracking-Area Objects" (page 78) explains how to use `NSTrackingArea` objects to manage mouse tracking and cursor updates within regions of views.
- "Handling Tablet Events" (page 85) describes how to handle events generated by moving and manipulating a stylus over a tablet device.

- "Text System Defaults and Key Bindings" (page 105) discusses the mechanism for binding key combinations to action messages and describes various defaults that can be applied to Cocoa's text system.

The appendix "Using Tracking-Area Objects" (page 78) covers the legacy API for mouse tracking and cursor updates. It explains how to set up tracking and cursor rectangles and handle the events that are subsequently generated when users move the mouse cursor into those areas.

See Also

The following ADC Reference Library documents are conceptually related to *Cocoa Event-Handling Guide*:

- *Cocoa Fundamentals Guide* (prerequisite reading)
- *View Programming Guide*
- *Text Input Management*

Because view objects often redraw themselves in response to events, it is also recommended that you peruse *Cocoa Drawing Guide*.

The following sample code projects in the ADC Reference Library include illustrative event-handling code:

- *BoingX* — mouse dragging, key events
- *CIAnnotation* — mouse clicks, mouse dragging
- *Cocoa OpenGL* — mouse clicks, key events
- *Color Sampler* — mouse clicks, mouse dragging
- *Cropped Image* — mouse clicks, mouse dragging
- *DragItemAround* — mouse clicks, mouse dragging, keyboard actions
- *FunkyOverlayWindow* — mouse tracking
- *ImageMapExample* — mouse dragging, mouse tracking
- *People* — key events
- *CircleView* — mouse clicks, mouse dragging
- *ClockControl* — responder related, keyboard actions
- *DotView* — mouse clicks
- *Rulers* — mouse dragging
- *SimpleStickies* — mouse clicks, mouse dragging

Event Architecture

The path taken by an event to the object in a Cocoa application that finally handles it can be a complicated one. This chapter traces the possible paths of events of various types and describes the mechanisms and architectural designs for handling events in the Application Kit.

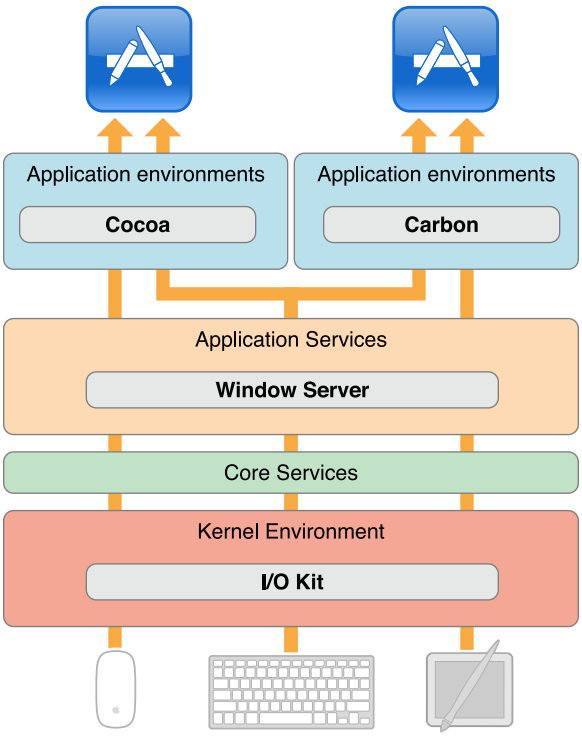
For further background, "About OS X App Design" in *Mac App Programming Guide* is recommended reading.

How an Event Enters a Cocoa Application

An event is a low-level record of a user action that is usually routed to the application in which the action occurred. A typical event in OS X originates when the user manipulates an input device attached to a computer system, such as a keyboard, mouse, or tablet stylus. When the user presses a key or clicks a button or moves a stylus, the device detects the action and initiates a transfer of data to the device driver associated with it. Through the I/O Kit, the device driver creates a low-level event, puts it in the window server's event queue,

and notifies the window server. The window server dispatches the event to the appropriate run-loop port of the target process. From there the event is forwarded to the event-handling mechanism appropriate to the application environment. Figure 1-1 depicts this event-delivery system.

Figure 1-1 The event stream



Note: Applications normally receive events from the keyboard and mouse only when they are in the foreground (that is, active). Although applications running in the background do not normally receive key and mouse events, low-level mechanisms called event taps make it possible for a background application to receive events and act upon them.

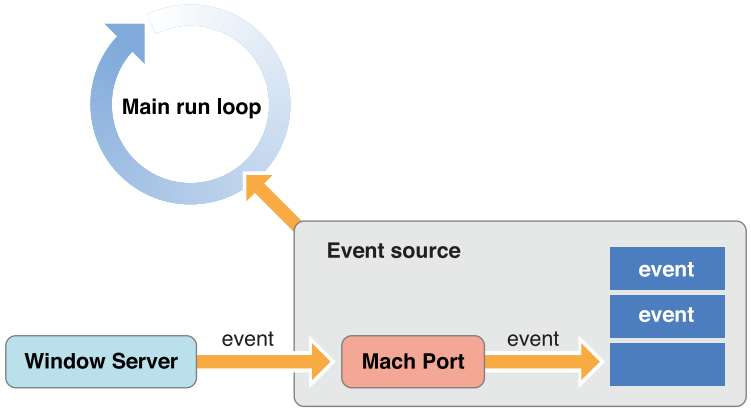
Before it dispatches an event to an application, the window server processes it in various ways; it time-stamps it, annotates it with the associated window and process port, and possibly performs other tasks as well. As an example, consider what happens when a user presses a key. The device driver translates the raw scan code into a virtual key code which it then passes off (along with other information about the key-press) to the window server in an event record. The window server has a translation facility that converts the virtual key code into a Unicode character.

In OS X, events are delivered as an asynchronous stream. This event stream proceeds "upward" (in an architectural sense) through the various levels of the system—the hardware to the window server to the Event Manager—until each event reaches its final destination: an application. As it passes through each subsystem, an event may change structure but it still identifies a specific user action.

> **Note:** Lower levels of the system trap and handle some events early in the event stream. These events are never routed to a Cocoa application. These events are generated by reserved keys or key combinations, such as the power and media-eject keys.

Every application has a mechanism specific to its environment for receiving events from the window server. For a Cocoa application, that mechanism is called the main event loop. A run loop, which in Cocoa is an `NSRunLoop` object, enables a process to receive input from various sources. By default, every thread in OS X has its own run loop, and the run loop of the main thread of a Cocoa application is called the main event loop. What especially distinguishes the main event loop is an input source called the event source, which is constructed when the global `NSApplication` object (`NSApp`) is initialized. The event source consists of a port for receiving events from the window server and a FIFO queue—the event queue—for holding those events until the application can process them, as shown in Figure 1-2.

Figure 1-2 The main event loop, with event source



A Cocoa application is event driven: It fetches an event from the queue, dispatches it to an appropriate object, and, after the event is handled, fetches the next event. With some exceptions (such as modal event loops) an application continues in this pattern until the user quits it. The following section, "Event Dispatch," describes how an application fetches and dispatches events.

Events delivered via the event source are not the only kinds of events that enter Cocoa applications. An application can also respond to Apple events, high-level interprocess events typically sent by other processes such as the Finder and Launch Services. For example, when users double-click an application icon to open the application or double-click a document to open the document, an Apple event is sent to the target application. An application also fetches Apple events from the queue but it does not convert them into `NSEvent` objects. Instead an Apple event is handled directly by an event handler. When an application launches, it automatically registers several event handlers for this purpose. For more on Apple events and event handlers, see *Apple Events Programming Guide*.

Event Dispatch

In the main event loop, the application object (`NSApp`) continuously gets the next (topmost) event in the event queue, converts it to an `NSEvent` object, and dispatches it toward its final destination. It performs this fetching of events by invoking the `nextEventMatchingMask:untilDate:inMode:dequeue:` method in a closed loop. When there are no events in the event queue, this method blocks, resuming only when there are more events to process.

After fetching and converting an event, `NSApp` performs the first stage of event dispatching in the `sendEvent:` method. In most cases `NSApp` merely forwards the event to the window in which the user action occurred by invoking the `sendEvent:` method of that `NSWindow` object. The window object then dispatches most events to the `NSView` object associated with the user action in an `NSResponder` message such as `mouseDown:` or `keyDown:`. An event message includes as its sole argument an `NSEvent` object describing the event.

The object receiving an event message differs slightly by type of event. For mouse and tablet events, the `NSWindow` object dispatches the event to the view over which the user pressed the mouse or stylus button. It dispatches most key events to the first responder of the key window. Figure 1-3 and Figure 1-4 illustrate these different general delivery paths. The destination view may decide not to handle the event, instead passing it up the responder chain (see "The Responder Chain" (page 23)).

Figure 1-3 Path of a mouse event

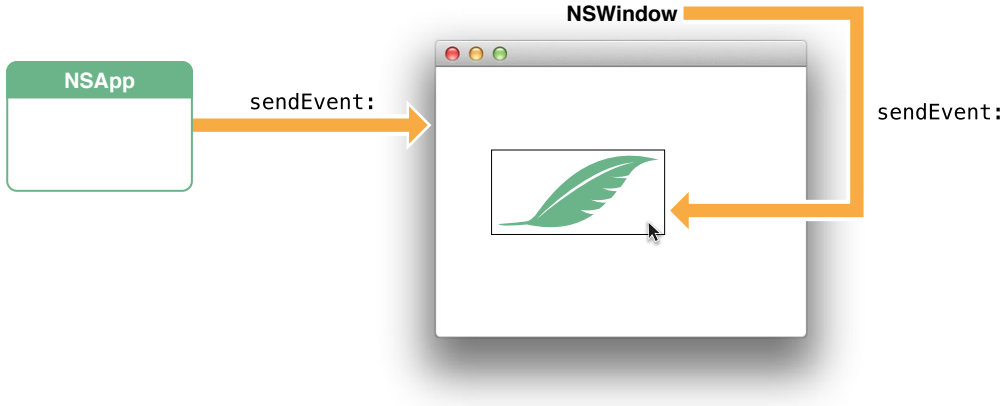
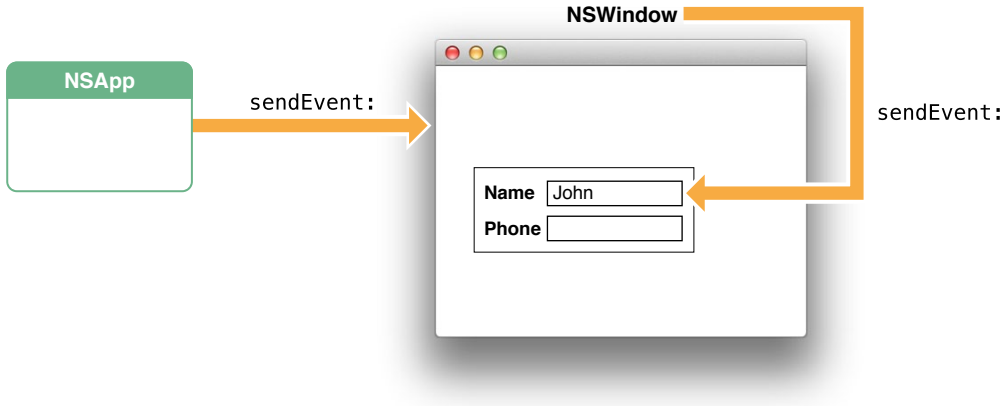


Figure 1-4 Path of a key event (character to insert)



In the preceding paragraph you might have noticed the use of qualifiers such as "in most cases" and "usually." The delivery of an event (and especially a key event) in Cocoa can take many different paths depending on the particular kind of event. Some events, many of which are defined by the Application Kit (type `NSAppKitDefined`), have to do with actions controlled by a window or the application object itself. Examples of these events are those related to activating, deactivating, hiding, and showing the application. `NSApp` filters out these events early in its dispatch routine and handles them itself.

The following sections describe the different paths of the events that can reach your view objects. For detailed information on these event types, read "Event Objects and Types" (page 27).

The Path of Mouse and Tablet Events

As noted above, an `NSWindow` object in its `sendEvent:` method forwards mouse events to the view over which the user action involving the mouse occurred. It identifies the view to receive the event by invoking the `NSView` method `hitTest:`, which returns the lowest descendant that contains the cursor location of the event (this is usually the topmost view displayed). The window object forwards the mouse event to this view by sending it a mouse-related `NSResponder` message specific to its exact type, such as `mouseDown:`, `mouseDragged:`, or `rightMouseUp:`, On (left) mouse-down events, the window object also asks the receiving view whether it is willing to become first responder for subsequent key events and action messages.

A view object can receive mouse events of three general types: mouse clicks, mouse drags, and mouse movements. Mouse-click events are further categorized—as specific `NSEventType` constants and `NSResponder` methods—by mouse button (left, right, or other) and direction of click (up or down). Mouse-dragged and mouse-up events are typically sent to the same view that received the most recent mouse-down event. Mouse-moved events are sent to the first responder. Mouse-down, mouse-dragged, mouse-up, and mouse-moved events can occur only in certain situations relative to other mouse events:

- Each mouse-up event must be preceded by a mouse-down event.
- Mouse-dragged events occur only between a mouse-down event and a mouse-up event.
- Mouse-moved events do not occur between a mouse-down and a mouse-up event.

Mouse-down events are sent when a user presses the mouse button while the cursor is over a view object. If the window containing the view is not the key window, the window becomes the key window and discards the mouse-down event. However, a view can circumvent this default behavior by overriding the `acceptsFirstMouse:` method of `NSView` to return `YES`.

Views automatically receive mouse-clicked and mouse-dragged events, but because mouse-moved events occur so often and can bog down the event queue, a view object must explicitly request its window to watch for them using the `NSWindow` method `setAcceptsMouseMovedEvents:`. Tracking rectangles, described in "Other Event Dispatching" (page 19), are a less expensive way of following the mouse's location.

In its implementation of an `NSResponder` mouse-event method, a subclass of `NSView` can interpret a mouse event as a cue to perform a certain action, such as sending a target-action message, selecting a graphic element, redrawing itself at a different location, and so on. Each event method includes as its sole parameter an `NSEvent` object from which the view can obtain information about the event. For example, the view can use the `locationInWindow` to locate the mouse cursor's hot spot in the coordinate system of the receiver's window. To convert it to the view's coordinate system, use `convertPoint:fromView:` with a `nil` view argument. From here, you can use `mouse:inRect:` to determine whether the click occurred in an interesting area.

Tablet events take a path to delivery to a view that is similar to that for mouse events. The `NSWindow` object representing the window in which the table event occurred forwards the event to the view under the cursor. However, there are two kinds of tablet events, proximity events and pointer events. The former are generally

native tablet events (of type `NSTabletProximity`) generated when the stylus moves into and out of proximity to the tablet. Tablet pointer events occur between proximity-entering and proximity-leaving tablet events and indicate such things as stylus direction, pressure, and button click. Pointer events are generally subtypes of mouse events. Refer to "Handling Tablet Events" (page 85) for more information.

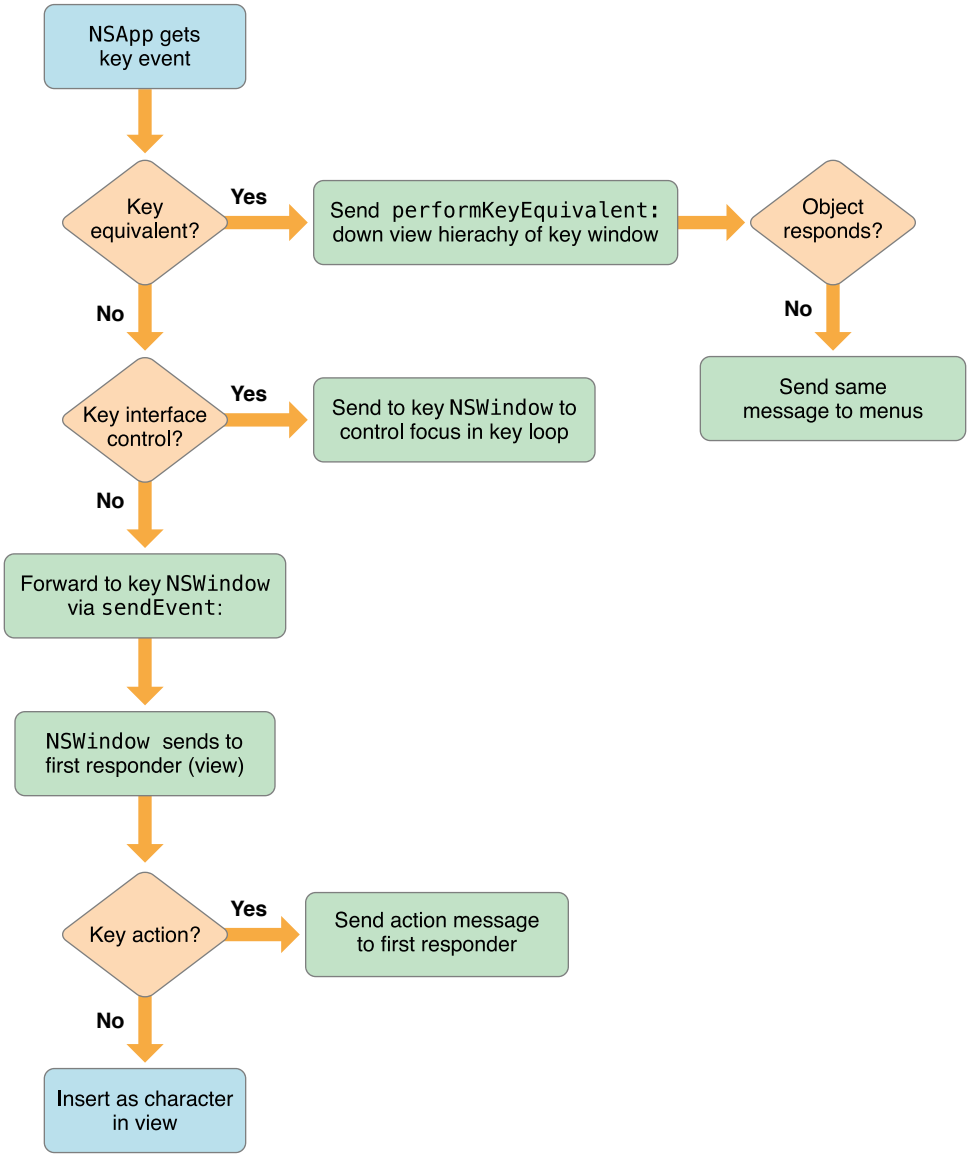
For the paths taken by mouse-tracking and cursor-update events, see "Other Event Dispatching" (page 19).

The Path of Key Events

Processing keyboard input is by far the most complex part of event dispatch. The Application Kit goes to great lengths to ease this process for you, and in fact handling the key events that get to your custom objects is fairly straightforward. However, a lot happens to those events on their way from the hardware to the responder chain. Of particular interest are the types of key events that arrive in a Cocoa application as `NSEvent` objects and the order and way these types of events are handled.

A Cocoa application evaluates each key event to determine what kind of key event it is and then handles in an appropriate way. The path a key event can take before it is handled can be quite long. Figure 1-5 shows these potential paths.

Figure 1-5 Possible path of a key event



The following list describes in detail the possible paths for key events, in the order in which an application evaluates each key event;

1. **Key equivalents**. A key equivalent is a key or key combination (usually a key modified by the Command key) that is bound typically to some menu item or control object in the application. Pressing the key combination simulates the action of clicking the control or choosing the menu item.

The application object handles key equivalents by going *down* the view hierarchy in the key window, sending each object a `performKeyEquivalent:` message until an object returns `YES`. If the message isn't handled by an object in the view hierarchy, `NSApp` then sends it to the menus in the menu bar. Some Cocoa classes, such as `NSButton`, `NSMenu`, `NSMatrix`, and `NSSavePanel` provide default implementations.

For more information, see "Handling Key Equivalents" (page 74).

2. **Keyboard interface control**. A keyboard interface control event manipulates the input focus among objects in a user interface. In the key window, `NSWindow` interprets certain keys as commands to move control to a different interface object, to simulate a mouse click on it, and so on. For example, pressing the Tab key moves input focus to the next object; Shift-Tab reverses the direction; pressing the space bar simulates a click on a button. The order of interface objects controlled through this mechanism is specified by a key view loop. You can set up the key view loop in Interface Builder and you can manipulate the key view loop programmatically through the `setNextKeyView:` and `nextKeyView` methods of `NSView`.

   For more information, see "Keyboard Interface Control" (page 75).

3. **Keyboard action**. Unlike the action messages that controls send to their targets (see "Action Messages" (page 19)), keyboard actions are commands (represented by methods defined by the `NSResponder` class) that are per-view functional interpretations of physical keystrokes (as identified by the constant returned by the `characters` method of `NSEvent`). In other words, keyboard actions are bound to physical keys through the key bindings mechanism described in "Text System Defaults and Key Bindings" (page 105). For example, `pageDown:`, `moveToBeginningOfLine:`, and `capitalizeWord:` are methods invoked by keyboard actions when the bound key is pressed. Such actions are sent to the first responder, and the methods handling these actions can be implemented in that view or in a superview further up the responder chain.

   See "Overriding the keyDown: Method" (page 67) for more information about the handling of keyboard actions.

4. **Character (or characters) for insertion as text**.

If the application object processes a key event and it turns out *not* to be a key equivalent or a key interface control event, it then sends it to the key window in a `sendEvent:` message. The window object invokes the `keyDown:` method in the first responder, from whence the key event travels *up* the responder chain until it is handled. At this point, the key event can be either one or more Unicode character to be inserted into a view's displayed text , a key or key combination to be specially interpreted, or a keyboard-action event.

See "Handling Key Events" (page 66) for more information about how key events are dispatched and handled.

Other Event Dispatching

An `NSWindow` object monitors tracking-rectangle events and dispatches these events directly to the owning object in `mouseEntered:` and `mouseExited:` messages. The owner is specified in the second parameter of the `NSTrackingArea` method `initWithRect:options:owner:userInfo:` and the `NSView` method`addTrackingRect:owner:userData:assumeInside:`. "Using Tracking-Area Objects" (page 78) describes how to set up tracking rectangles and handle the related events.

Periodic events (type `NSPeriodic`) are generated by the application at a specified frequency and placed in the event queue. However, unlike most other types of events, periodic events aren't dispatched using the `sendEvent:` mechanism of `NSApplication` and `NSWindow`. Instead the object registering for the periodic events typically retrieves them in a modal event loop using the `nextEventMatchingMask:untilDate:inMode:dequeue:` method. See "Other Types of Events" (page 38) for more information about periodic events.

Action Messages

The discussion so far has focused on event messages: messages arising from an device-related event such as a mouse click or a key-press. The Application Kit sends an event message of the appropriate form—for example, `mouseDown:` and `keyDown:`—to an `NSResponder` object for handling.

But `NSResponder` objects are also expected to handle another kind of message: action messages. Actions are commands that objects, usually `NSControl` or `NSMenu` objects, give to the application object to dispatch as messages to a particular target or to any target that's willing to respond to them. The methods invoked by action messages have a specific signature: a single parameter holding a reference to the object initiating the action message; by convention, the name of this parameter is `sender`. For example,

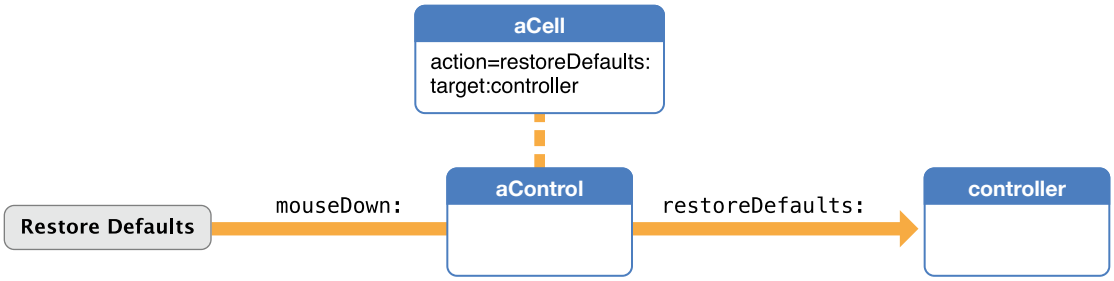
```
- (void)moveToEndOfLine:(id)sender; // from NSResponder.h
```

Event and action methods are dispatched in different ways, by different methods. Nearly all events enter an application from the window server and are dispatched automatically by the `sendEvent:` method of `NSApplication`. Action messages, on the other hand, are dispatched by the `sendAction:to:from:` method of the global application object (`NSApp`) to their proper destinations.

> **Note:** A major difference between event messages and action messages is the different paths they can possibly take up the responder chain. See "The Responder Chain" (page 23) for details.

As illustrated in Figure 1-6, action messages are generally sent as a secondary effect of an event message. When a user clicks a control object such as a button, two event messages (`mouseDown:` and `mouseUp:`) are sent as a result. The control and its associated cell handle the `mouseUp:` message (in part) by sending the application object a `sendAction:to:from:` message. The first argument is the selector identifying the action method to invoke. The second is the intended recipient of the message, called the target, which can be `nil`. The final argument is usually the object invoking `sendAction:to:from:`, thus indicating which object initiated the action message. The target of an action message can send messages back to `sender` to get further information. A similar sequence occurs for menus and menu items. For more on the architecture of controls and cells (and menus and menu items) see "The Core App Design" in *Mac App Programming Guide*.

Figure 1-6 From event message to action message



The target of an action message is handled by the Application Kit in a special way. If the intended target isn't `nil`, the action is simply sent directly to that object; this is called a targeted action message. In the case of an untargeted action message (that is, the target parameter is `nil`), `sendAction:to:from:` searches up the full responder chain (starting with the first responder) for an object that implements the action method specified. If it finds one, it sends the message to that object with the initiator of the action message as the sole argument. The receiver of the action message can then query the sender for additional information. You can find the recipient of an untargeted action message without actually sending the message using `targetForAction:`.

Event messages form a well-known set, so `NSResponder` provides declarations and default implementations for all of them. Most action messages, however, are defined by custom classes and can't be predicted. However, `NSResponder` does declare a number of keyboard action methods, such as `pageDown:`, `moveToBeginningOfDocument:`, and `cancelOperation:`. These action methods are typically bound to specific keys using the key-bindings mechanism and are meant to perform cursor movement, text operations, and similar operations.

A more general mechanism of action-message dispatch is provided by the `NSResponder` method `tryToPerform:with:`. This method checks the receiver to see if it responds to the selector provided, if so invoking the message. If not, it sends `tryToPerform:with:` to its next responder. `NSWindow` and `NSApplication` override this method to include their delegates, but they don't link individual responder chains in the way that the `sendAction:to:from:` method does. Similar to `tryToPerform:with:` is `doCommandBySelector:`, which takes a method selector and tries to find a responder that implements it. If none is found, the method causes the hardware to beep.

Warning: Although `NSResponder` declares a number of action messages, it doesn't actually implement them. You should never send an action message directly to a responder object of an unknown class. Always use the `NSApplication` method `sendAction:to:from:`, the `NSResponder` methods `tryToPerform:with:` or `doCommandBySelector:`, or check that the target responds using the `NSObject` method `respondsToSelector:`.

Responders

A responder is an object that can receive events, either directly or through the responder chain, by virtue of its inheritance from the `NSResponder` class. `NSApplication`, `NSWindow`, `NSDrawer`, `NSWindowController`, `NSView` and the many descendants of these classes in the Application Kit inherit from `NSResponder`. This class defines the programmatic interface for the reception of event messages and many action messages. It also defines the general structure of responder behavior. Within the responder chain there is a first responder and a sequence of next responders

For more on the responder chain, see "The Responder Chain" (page 23).

First Responders

A first responder is typically a user-interface object that the user selects or activates with the mouse or keyboard. It is usually the first object in a responder chain to receive an event or action message. An `NSWindow` object's first responder is initially itself; however, you can set, programmatically and in Interface Builder, the object that is made first responder when the window is first placed on-screen.

When an `NSWindow` object receives a mouse-down event, it automatically tries to make the `NSView` object under the event the first responder. It does so by asking the view whether it wants to become first responder, using the `acceptsFirstResponder` method defined by this class. This method returns `NO` by default; responder subclasses that need to be first responder must override it to return `YES`. The `acceptsFirstResponder` method is also invoked when the user changes the first responder through the keyboard interface control feature.

You can programmatically change the first responder by sending `makeFirstResponder:` to an `NSWindow` object. This message initiates a kind of protocol in which one object loses its first responder status and another gains it. See "Setting the First Responder" (page 48) for further information.

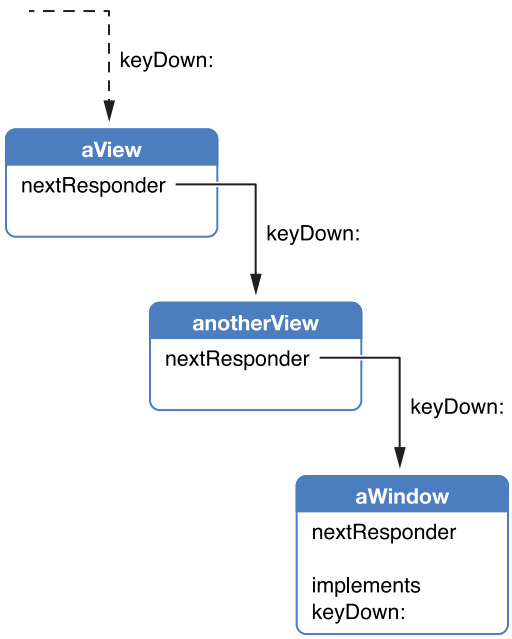
An `NSPanel` object presents a variation of first-responder behavior that permits panels to present a user interface that doesn't take away key focus from the main window. If the panel object representing an inactive window and returning `YES` from `becomesKeyOnlyIfNeeded` receives a mouse-down event, it attempts to make the view object under the mouse pointer the first responder, but *only* if that object returns `YES` in `acceptsFirstResponder` and `needsPanelToBecomeKey`.

Mouse-moved events (type `NSMouseMoved`) are always sent to the first responder, not to the view under the mouse.

Next Responders

Every responder object has a built-in capability for getting the next responder up the responder chain. The `nextResponder` method, which returns this object, is the essential mechanism of the responder chain. Figure 1-7 shows the sequence of next responders.

Figure 1-7 The chain of next responders



A view's next responder is always its superview—most of the responder chain, in fact, comprises the views from a window's first responder up to its content view. When you create a window or add subviews to existing views, either programmatically or in Interface Builder, the Application Kit automatically hooks up the next responders in the responder chain. The `addSubview:` method of `NSView` automatically sets the receiver as

the new subview's superview. You should never send `setNextResponder:` to an `NSView` object. You can safely add responders to the top end of a window's responder chain—the `NSWindow` object itself if it has no delegate or, if it has a delegate, after the delegate.

The Responder Chain

The responder chain is a linked series of responder objects to which an event or action message is applied. When a given responder object doesn't handle a particular message, the object passes the message to its successor in the chain (that is, its next responder). This allows responder objects to delegate responsibility for handling the message to other, typically higher-level objects. The Application Kit automatically constructs the responder chain as described below, but you can insert custom objects into parts of it using the `NSResponder` method `setNextResponder:` and you can examine it (or traverse it) with `nextResponder`.

An application can contain any number of responder chains, but only one is active at any given time. The responder chain is different for event messages and action messages, as described in the following sections.

Responder Chain for Event Messages

Nearly all event messages apply to a single window's responder chain—the window in which the associated user event occurred. The default responder chain for event messages begins with the view that the `NSWindow` object first delivers the message to. The default responder chain for a key event message begins with the first responder in a window; the default responder chain for a mouse or tablet event begins with the view on which the user event occurred. From there the event, if not handled, proceeds up the view hierarchy to the `NSWindow` object representing the window itself. The first responder is typically the "selected" view object within the window, and its next responder is its containing view (also called its superview), and so on up to the `NSWindow` object. If an `NSWindowController` object is managing the window, it becomes the final next responder. You can insert other responders between `NSView` objects and even above the `NSWindow` object near the top of the chain. These inserted responders receive both event and action messages. If no object is found to handle the event, the last responder in the chain invokes `noResponderFor:`, which for a key-down event simply beeps. Event-handling objects (subclasses of `NSWindow` and `NSView`) can override this method to perform additional steps as needed.

Responder Chain for Action Messages

For action messages, the Application Kit constructs a more elaborate responder chain that varies according to two factors:

- Whether the application is based on the document architecture and, if it isn't, whether it uses `NSWindowController` objects for its windows
- Whether the application is currently displaying a key window as well as a main window

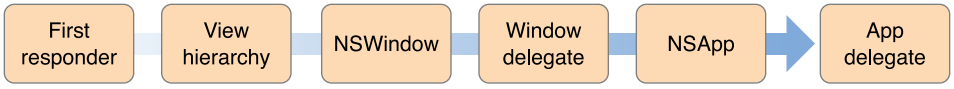
Action messages have a more elaborate responder chain than do event messages because actions require a more flexible runtime mechanism for determining their targets. They are not restricted to a single window, as are event messages.

The simplest case is an active non-document-based window that has no associated panel or secondary window displayed—in other words, a main window that is also the key window. In this case, the responder chain is the following:

1. The main window's first responder and the successive responder objects up the view hierarchy
2. The main window itself
3. The main window's delegate (which need not inherit from `NSResponder`)
4. The application object, `NSApp`
5. The application object's delegate (which need not inherit from `NSResponder`)

This chain is shown graphically in Figure 1-8.

Figure 1-8 Responder chain of a non-document-based application for action messages



As this sequence indicates, the `NSWindow` object and the `NSApplication` object give their delegates a chance to handle action messages as though they were responders, even though a delegate isn't formally in the responder chain (that is, a `nextResponder` message to a window or application object doesn't return the delegate).

When an application is displaying both a main window and a key window, the responder chains of both windows can be involved in an action message. As explained in "Window Layering and Types of Windows", the main window is the frontmost document or application window. Often main windows also have key status, meaning they are the current focus of user input. But a main window can have a secondary window or panel associated with it, such as the Find panel or a Info window showing details of a selection in the document window. When this secondary window is the focus of user input, then it is the key window.

When an application has a main window *and* a separate key window displayed, the responder chain of the key window gets first crack at action messages, and the responder chain of the main window follows. The full responder chain comprises these responders and delegates:

1. The key window's first responder and the successive responder objects up the view hierarchy
2. The key window itself
3. The key window's delegate (which need not inherit from `NSResponder`)

4. The main window's first responder and the successive responder objects up the view hierarchy
5. The main window itself
6. The main window's delegate (which need not inherit from `NSResponder`)
7. The application object, `NSApp`
8. The application object's delegate (which need not inherit from `NSResponder`)

As you can see, the responder chains for the key window and the main window are identical with the global application object and its delegate being the responders at the end of the main window's responder chain. This design is true for the responder chains of the other kinds of applications: those based on the document architecture and those that use an `NSWindowController` object for window management. In the latter case, the default main-window responder chain consists of the following responders and delegates:

1. The main window's first responder and the successive responder objects up the view hierarchy
2. The main window itself
3. The window's `NSWindowController` object (which inherits from `NSResponder`)
4. The main window's delegate
5. The application object, `NSApp`
6. The application object's delegate

Figure 1-9 shows the responder chain of non-document-based application that uses an NSWindowController object.

Figure 1-9 Responder chain of a non-document application with an `NSWindowController` object (action messages)

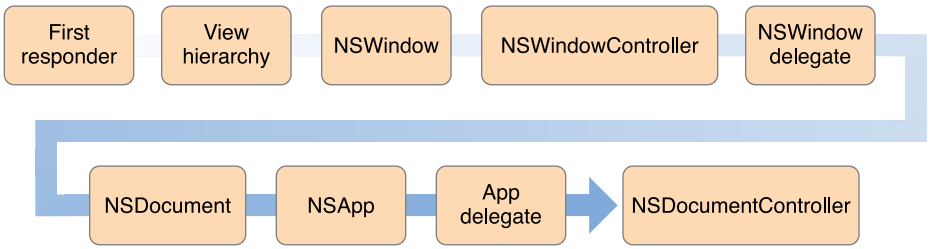
For document-based applications, the default responder chain for the main window consists of the following responders and delegates:

1. The main window's first responder and the successive responder objects up the view hierarchy
2. The main window itself
3. The window's `NSWindowController` object (which inherits from `NSResponder`)
4. The main window's delegate.
5. The `NSDocument` object (if different from the main window's delegate)
6. The application object, `NSApp`

7. The application object's delegate
8. The application's document controller (an `NSDocumentController` object, which does not inherit from `NSResponder`)

Figure 1-10 shows the responder chain of a document-based application.

Figure 1-10 Responder chain of a document-based application for action messages



Other Uses

The responder chain is used by three other mechanisms in the Application Kit:

- **Automatic menu item and toolbar item enabling**: In automatically enabling and disabling a menu item with a `nil` target, an `NSMenu` searches different responder chains depending on whether the menu object represents the application menu or a context menu. For the application menu, `NSMenu` consults the full responder chain—that is, first key, then main window—to find an object that implements the menu item's action method and (if it implements it) returns `YES` from `validateMenuItem:`. For a context menu, the search is restricted to the responder chain of the window in which the context menu was displayed, starting with the associated view.

  Enabling and disabling of toolbar items makes use of the responder chain in a fashion identical to that of menu items. In this case, the key validation method is `validateToolbarItem:`.

  For more on automatic menu-item enabling, see "Enabling Menu Items"; for more on validation of toolbar items, see "Validating Toolbar Items".

- **Services eligibility**: Similarly, the Services facility passes `validRequestorForSendType:returnType:` messages along the full responder chain to check for objects that are eligible for services offered by other applications.

  For further information, see *Services Implementation Guide*.

- **Error presentation**: The Application Kit uses a modified version of the responder chain for error handling and error presentation, centered upon the `NSResponder` methods `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` and `presentError:`.

  For more information on the error-responder chain, see *Error Handling Programming Guide*.

Event Objects and Types

Almost all events in a Cocoa application are represented by objects of the `NSEvent` class. (Exceptions include Apple events, notifications, and similar items.) Each `NSEvent` object more narrowly represents a particular type of event, each with its own requirements for handling. The following sections describe the characteristics of an `NSEvent` object and the possible event types.

NSEvent Objects

An `NSEvent` object—or, simply, an event object—contains pertinent information about an input action such as a mouse click or a key press. It stores such details as where the mouse was located or which character was typed. As described in "How an Event Enters a Cocoa Application" (page 10), the window server associates each user action with a window and reports the event (in a lower-level form) to the application that created the window. The application temporarily places each event in a buffer called the event queue. When the application is ready to process an event, the application object (`NSApp`) takes one from the queue (usually the topmost one in the queue) and converts it to an `NSEvent` object before dispatching it to the appropriate objects in an application.

Responder objects of an application receive the currently dispatched event object through the parameter of an event method declared by the `NSResponder` class (such as `mouseDown:`). In addition, other methods of the Application Kit let any object retrieve the current event or fetch the next event (or next event of a specific type) from the event queue. See "Event Objects in Methods of Other Classes" (page 29) for more information about these methods.

Attributes of an Event Object

An `NSEvent` object is largely a read-only repository of information related to a specific event. Most methods of the `NSEvent` class are accessor methods for getting the values of event attributes. (`NSEvent` has no corresponding "setter" accessor methods, although you can specify certain attributes when creating an event object using various class factory methods.) An object such as a responder typically uses the accessor methods to get the details of an event and thus know how to handle it.

Some NSEvent attributes (and their corresponding accessor methods) are common to all types of events while others are specific to certain types of events. For example, the clickCount method pertains only to mouse events and the characters method pertains only to key events. Tablet events have a number of accessor methods that apply only to them. Some of the more important accessor methods of NSEvent are the following:

type
: The type of event; see "Event Types" (page 30).

window
: The NSWindow object representing the window in which the event occurred. With windowNumber you can also obtain the number assigned by the window server to the window. Most but not all events are associated with a window; when no window is associated, window returns nil.

locationInWindow
: The location of the event within the window's base coordinate system.

modifierFlags
: An indication of which modifier keys (Command, Control, Shift, and so on) the user held down while the event occurred.

characters
: The Unicode characters generated by a key event. You can also use charactersIgnoringModifiers to obtain the key-event characters minus those generated by modifier keys.

timestamp
: The time the event occurred (in seconds since system startup).

clickCount
: For mouse events within a certain time threshold, the number of clicks associated with a particular event. (This enables the detection of double- or triple-clicking.)

NSEvent Class Methods

Though you rarely need do so, you can create an event object from scratch and either insert it into the event queue for distribution or send it directly to its destination in an event message. The NSEvent class includes class methods for creating event objects of a specific type; for example, for creating a mouse-type event object, you can use the class method mouseEventWithType:location:modifierFlags:timestamp:windowNumber:context:eventNumber:clickCount:pressure:. You add event objects to the event queue by invoking the NSWindow method postEvent:atStart: or the identically named method of the NSApplication class.

Another `NSEvent` class method, `mouseLocation`, returns the current location of the mouse. It differs in a few important respects from the instance method `locationInWindow`. Being a class method, it doesn't require an event object to send the message to; it returns the location in screen coordinates rather than base (window) coordinates; and it returns the current mouse location, which might be different from that of the current or any pending mouse event.

> **Note:** OS X v10.6 introduced several class methods that allow access to some system preferences for keyboard and mouse. The `keyRepeatDelay` class method returns a time value representing the period a key must be pressed to generate the first key-repeat event. The `keyRepeatInterval` class method returns the time between subsequent key-repeat events. For mouse events there is a `doubleClickInterval` class method which returns the period in which a second click must occur to be considered a double-click.
>
> In OS X v10.6 there are two other class methods that a caller outside of the event stream can use to learn which modifiers keys are pressed (`modifierFlags`) and which mouse buttons are pressed (`pressedMouseButtons`).

Event Objects in Methods of Other Classes

`NSEvent` objects are scattered throughout the Application Kit. For example, the classes `NSCell`, `NSCursor`, `NSClipView`, `NSMenu`, `NSSlider`, and `NSTableView` all have methods with event objects as return values or parameters. However, a few Application Kit methods that deal with event objects are particularly important.

Although most events are distributed automatically through the responder chain, sometimes an object needs to retrieve events explicitly—for example, while mouse tracking. Both `NSWindow` and `NSApplication` define the method `nextEventMatchingMask:untilDate:inMode:dequeue:`, which allows an object to retrieve events of specific types from the event queue.

`NSApplication` and `NSWindow` also both define the `currentEvent` method, which fetches the last event object retrieved from the event queue. These methods are a great convenience because they enable any object in an application to learn about the event currently being handled in the main event loop.

Finally, both `NSWindow` and `NSApplication` define the `sendEvent:` method. The implementations of these methods are critical for event dispatch. Because these methods are funnel points for events in an application, you may override them in certain circumstances to learn about events earlier in the event stream or to augment or modify the native event-dispatch behavior. "Event Dispatch" talks about the role played by the `sendEvent:` methods.

Event Types

The `type` method of `NSEvent` returns an `NSEventType` value that identifies the sort of event. The type of an event determines to a large extent how it is to be handled. The different types of events fall into six groups:

- Mouse events
- key events
- Tracking-rectangle and cursor-update events
- Tablet events
- Periodic events
- Other events

Some of these groups comprise several `NSEventType` constants, others only one. Some event types might have subtypes, which are described in the following sections. `NSEventType` constants are declared in `NSEvent.h`.

The `NSApplication` and `NSWindow` methods that allow you to selectively fetch and discard events from the event queue—`nextEventMatchingMask:untilDate:inMode:dequeue:` and `discardEventsMatchingMask:beforeEvent:`—take one or more event-type mask constants in the first parameter. These constants are also declared in `NSEvent.h`.

Mouse Events

Mouse events are generated by changes in the state of the mouse buttons and by changes in the position of the mouse cursor on the screen. They fall into two subcategories, one related to mouse clicks and movements and the other related to mouse tracking and cursor updates.

Events Related to Mouse Clicks and Movements

The larger category of mouse events includes those where the mouse button is pressed or released and where the mouse is moved without being tracked. It consists of the following mouse-event types corresponding to the specified user actions:

`NSLeftMouseDown`, `NSLeftMouseUp`, `NSRightMouseDown`, `NSRightMouseUp`, `NSOtherMouseDown`, `NSOtherMouseUp`

> The user clicked a mouse button. Constants with “MouseDown” in their names mean the user pressed the button; “MouseUp” means the user released it. If the mouse has just one button, only left mouse events are generated. By sending a `clickCount` message to the event, you can determine whether the

mouse event was a single click, double click, and so on. A mouse with more than two buttons can generate the "OtherMouse" events.

`NSLeftMouseDragged`, `NSRightMouseDragged`, `NSOtherMouseDragged`
: The user dragged the mouse. More specifically, the user moved the mouse while pressing one or more buttons. `NSLeftMouseDragged` events are generated when the mouse is moved with its left mouse button down or with both buttons down, `NSRightMouseDragged` events are generated when the mouse is moved with just the right button down, and `NSOtherMouseDragged` when the device has more than two buttons. A mouse with a single button generates only left mouse-dragged events. A series of mouse-dragged events is always preceded by a mouse-down event and followed by a mouse-up event.

`NSMouseMoved`
: The user moved the mouse without holding down either mouse button. Mouse-moved events are normally not tracked, as they quickly flood the event queue; use the `NSWindow` method `setAcceptsMouseMovedEvents:` to turn on tracking of mouse movements.

`NSScrollWheel`
: The user manipulated the mouse's scroll wheel. Use the `NSEvent` methods `deltaX`, `deltaY`, and `deltaZ` to find out how much it moved. If the mouse has no scroll wheel, this event is never generated.

Note: Beginning with OS X v10.5, `NSScrollWheel` events are sent to the window under the mouse, whether the window is active or inactive. In earlier versions of the operating system, scroll-wheel events are sent to the window under the mouse only if that window has key focus (with the exception of utility windows, which receive those events even if they are inactive).

Mouse-dragged and mouse-moved events are generated repeatedly as long as the user keeps moving the mouse. If the mouse is stationary, neither type of event is generated until the mouse moves again.

Important: The Application Kit does not provide any default handling of events generated by the third button of a three-button mouse (types `NSOtherMouseDown`, `NSOtherMouseDragged`, and `NSOtherMouseUp`).

See "Handling Mouse Events" (page 51) for more information on mouse events.

Mouse Tracking Events

Because following the mouse's movements precisely is an expensive operation, the Application Kit provides a less intensive mechanism for tracking the location of the mouse. It does this by allowing the application to define regions of a window, called tracking rectangles, that generate events when the cursor enters or leaves them. The event types are `NSMouseEntered` and `NSMouseExited` and they're generated when the application has asked the window server to set a tracking rectangle in a window, typically by using `NSTrackingArea`

objects or the NSView method addTrackingRect:owner:userData:assumeInside:. A window can have any number of tracking rectangles; the NSEvent method trackingNumber identifies the rectangle that triggered the event.

A special kind of tracking event is the NSCursorUpdate event. This type is used to implement the cursor-rectangle mechanism of the NSView class. An NSCursorUpdate event is generated when the cursor has crossed the boundary of a predefined rectangular area. NSApplication typically handles NSCursorUpdate events and does not dispatch them.

See "Using Tracking-Area Objects" (page 78) for more information.

Key Events

Among the most common events sent to an application are direct reports of the user's keyboard actions, identified by these NSEventType constants:

NSKeyDown
: The user generated a character or characters by pressing a key.

NSKeyUp
: The user released a key. This event is always preceded by a NSKeyDown event.

NSFlagsChanged
: The user pressed or released a modifier key, or turned Caps Lock on or off.

Of these, key-down events are the most useful to an application. When the type of an event is NSKeyDown, the next step is typically to get the characters generated by the key-down using the characters method.

Key-up events are used less frequently since they follow almost automatically when there's been a key-down event. And because the NSEvent modifierFlags method returns the state of the modifier keys regardless of the type of event, applications normally don't need to receive flags-changed events; they're useful only for applications that have to keep track of the state of these keys at all times.

Some key presses generate key events that do not represent characters to be inserted as text. Instead they represent key equivalents, keyboard interface control commands, or keyboard actions. Key equivalents and keyboard interface control commands are typically handled by the application object and do not invoke the NSResponder method associated with NSKeyDown events, keyDown:. See "The Path of Key Events" (page 16) for more information.

For more information on key events, see "Handling Key Events" (page 66).

Tablet Events

Tablet devices generate low-level events that a Cocoa application receives as `NSEvent` objects. The following sections describe tablet devices, the characteristics of tablet events, and how the Application Kit supports tablet events.

Important: Tablet events are available in OS X v10.4 and later versions of the operating system.

Overview of Tablet Devices

A tablet with a stylus is an input device that generates more accurate and detailed data than does a mouse. It enables a user to draw, write, or make selections by manipulating the stylus over a surface (the tablet); an application can then capture and process those movements, reflecting them in its user interface. The tablet is generally a USB device connected to a computer system and the stylus is a wireless transducer. A signal is sent from the tablet to the transducer, which then sends a signal back to the tablet. The tablet uses this signal to determine the position of the transducer on the tablet. The stylus actually can be any pointing device, such as a pen, an airbrush, or even a puck.

In addition to the stylus location at any given moment, a stylus transducer can report many other pieces of data, such as the tilt of a pen, the rotation of a puck, and the pressure applied to the stylus. Pressure is particularly important because, with just this small piece of data, a user can tell an application to vary the thickness of a line being drawn, or its opacity, or its color. Some stylus devices also have buttons that can furnish an application with additional information.

OS X supports tablet devices from several manufacturers. Some of these tablets can respond to multiple pointing devices on their surfaces at the same time.

Types of Tablet Events

There are two types of tablet events in the Application Kit: proximity events and pointer events. The following sections describe what they are, how they are related to each other, and the sequence of proximity and pointer events in a typical tablet session.

Proximity Events

A proximity event is an event that a tablet device generates when a pointing transducer (for example, a stylus) comes near or moves away from the tablet surface. It indicates the start or the end of a related series of pointer events (a session). A proximity event is an `NSEvent` object of type `NSTabletProximity`. An application can determine whether a tablet-pointer session is beginning or ending by sending `isEnteringProximity` to the event object.

The main purpose of a proximity event is to provide an application with a set of identifiers for associating items of tablet hardware—entire tablet devices or individual transducers—with the current pointing session. A proximity tablet event can also furnish an application with information about the capabilities of a specific device.

A proximity-type tablet event carries with it a set of identifiers and device attributes that an application can fetch with accessor methods. These include the following:

- Device ID—The main identifier of a tablet device, used to associate pointer-type events with proximity events. All tablet pointer events in a session have the same device ID.

  Accessor: `deviceID`

- Pointing devices—To help an application distinguish among pointing devices, you can ask a proximity event for the device's serial number, its type (for example, pen or eraser), and, for tablets that support multiple concurrent pointing devices, its device ID.

  Accessors: `pointingDeviceSerialNumber`, `pointingDeviceType`, `pointingDeviceID`

- Tablets—You can ask a proximity event for a tablet's identifier (which is its USB model number) and, if there are multiple tablets connected to a system, its system tablet ID.

  Accessors: `tabletID`, `systemTabletID`

- Vendor information—An `NSEvent` object with a proximity type may contain an identifier of the vendor and an identifier of a pointing device within a vendor's selection of devices.

  Accessors: `vendorID`, `vendorPointingDeviceType`

- Capabilities—A bit mask whose set bits indicate the capabilities of a tablet device. It is vendor-specific.

  Accessor: `capabilityMask`

Generally, when an application receives a proximity event, it stores the device ID and any other identifier that is needed to distinguish among various items of tablet hardware involved in the session. It then refers to these identifiers when handling pointer events to ensure it is processing the right events. An application can also extract device information from proximity events (for example, tablet capabilities or pointer type) and use this information to configure how it deals with pointer events.

Pointer Events

A pointer event is an event that a tablet device generates after a stylus has entered proximity of the tablet. It indicates a change in the state of the transducer. For example, if the user moves a stylus transducer over the surface of the tablet or increases pressure or tilts the pointing device, a pointer event is generated. A pointer event is an `NSEvent` object of type `NSTabletPoint` or an object representing a mouse-down, mouse-dragged, or mouse-up event with a subtype of `NSTabletPointEventSubtype`.

Applications generally use pointer events for drawing or user-interface manipulation. Although you can obtain the absolute three-dimensional coordinates of the current pointer location, these coordinates are in full tablet resolution and require you to scale them to the screen location. It is much simpler to use the `NSEvent` instance method `locationInWindow` or the class method `mouseLocation`.

In addition to pointer location, the information obtainable from pointer events include the following:

- Pressure, as a value between 0.0 and 1.0; you might use the pressure attribute to set the opacity of a color.

  Accessor: `pressure`

- Rotation, in degrees; you might use the rotation attribute to simulate a calligraphy pen.

  Accessor: `rotation`

- Tilt, an `NSPoint` structure, with both axes ranging from -1 to 1; you might use the tilt attribute to supply different colors, depending on the angle and direction of tilt.

  Accessor: `tilt`

- Tangential pressure, as a value between -1.0 and 1.0 (only on certain devices).

  Accessor: `tangentialPressure`

- Button number of transducer pressed.

  Accessor: `buttonMask`

- Vendor-defined data.

  Accessor: `vendorDefined`

Sequence of Tablet Events

A tablet proximity event signals the start of a series of related tablet pointer event and a subsequent proximity event signals the end of the series. These two proximity events thus provide a kind of frame for processing the pointer events in a session. The first proximity event is generated when a pointing device comes near a tablet surface; the second is generated when the same pointing device leaves the proximity of the tablet.

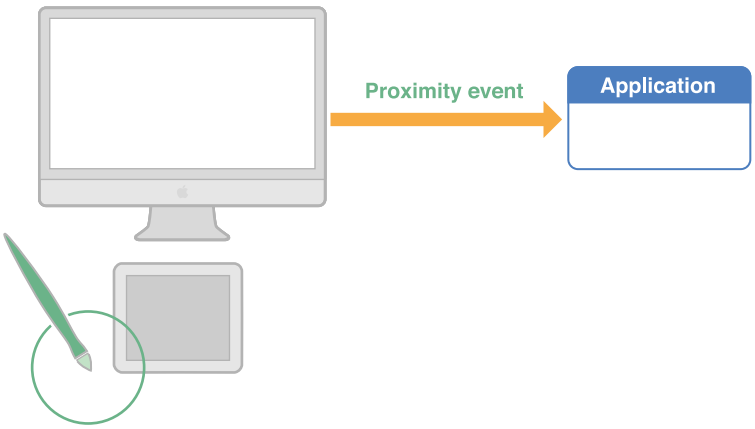
> **Note:** You can determine whether a proximity event is for a pointing device entering proximity or leaving proximity by sending the event object an `isEnteringProximity` message.

The sequence of proximity and pointer events have special significance to the application handling the tablet events. A tablet event generated by a pointing device that comes near the tablet lets the tablet application know that it should store identifiers and set configuration variables for the upcoming tablet session. The application processes the pointer events until it receivers the second proximity event, which tells it that those identifiers and configurations are no longer operative. A tablet session is thus a sequence of pointer events that are associated with a specific pair of proximity events.

The relationship between proximity and pointer events is simple and clear as long as only one pointing device is in play. But there can be multiple pointing devices on a tablet surface at one time, or more than one tablet devices can be connected to a system. In this case the application must store the identifiers it receivers in all initial proximity events and use those identifiers to differentiate among various series of pointer events.

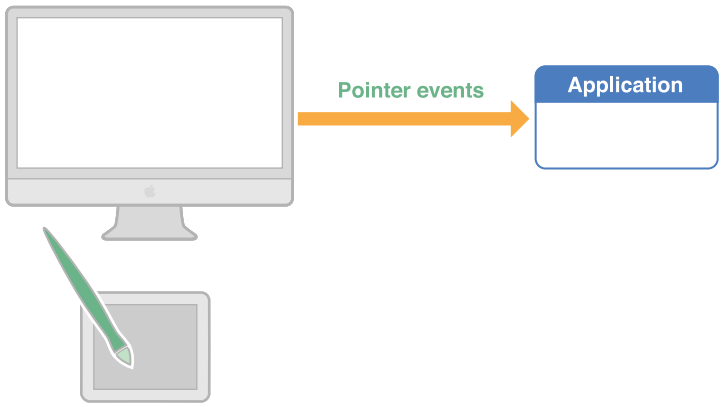
Take, for example, a tablet device that supports multiple pointing device on the tablet surface at one time. One pointing device might be a stylus for line drawing; another pointing device might apply an airbrush painting effect. As Figure 2-1 illustrates, a proximity event is generated with the drawing stylus comes close to the tablet surface.

Figure 2-1 Pointer A coming near the tablet, generating proximity event



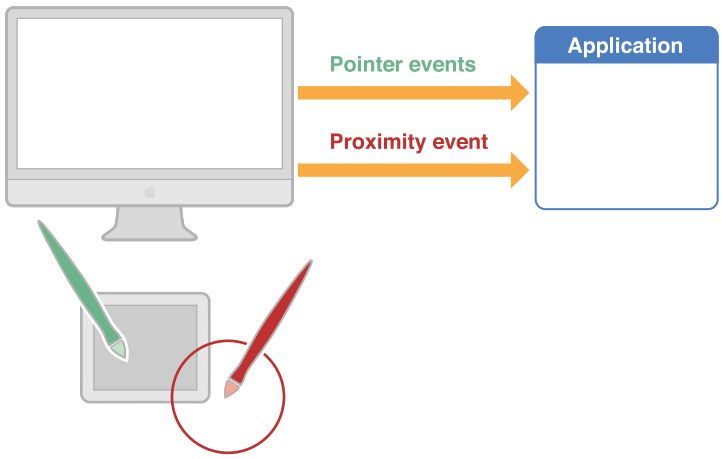
When handling this proximity event, the application stores the device ID of the tablet device, the device ID of the pointing device, and perhaps also the serial number of the pointing device and its type. Until it receives the next proximity event with these same identifiers, it processes all pointer events it receives for the pointing device—in this case, drawing lines (Figure 2-2).

Figure 2-2 Application receiving pointer events



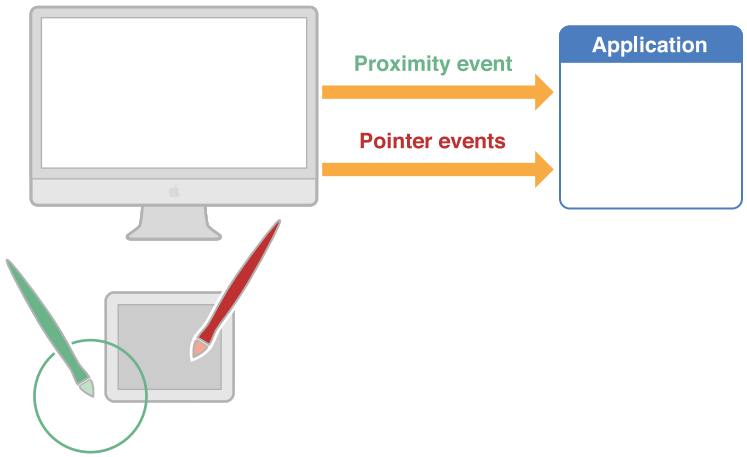
Now the airbrush pointing device moves onto the surface of the tablet, generating another proximity event (Figure 2-3). Because this event carries with it different identifiers, the application knows its not the terminating proximity event for the first tablet session. Instead it announces an impending series of pointer events for another pointing device. So the application stores the identifiers and configuration information for this session.

Figure 2-3 Pointer B coming near the tablet, generating proximity event



For a period, the tablet application is processing two different series of pointer events, using the stored identifiers to distinguish between the two. Then, as depicted in Figure 2-4, the drawing pointing device moves away from the tablet surface. This action generates a proximity event. The application examines the event and sees that the identifiers for the pointing device are the same as what it initially stored for the line-drawing device. It nullifies the stored identifiers, bringing to a close the initial series of pointer events.

Figure 2-4 Pointer A leaving the tablet surface, generating proximity event



Other Types of Events

The Application Kit defines several minor event types. Some of these are rarely used, but they are available if you ever find need for them.

A periodic event (type `NSPeriodic`) simply notifies an application that a certain time interval has elapsed. Periodic events are particularly useful in situations where input events aren't generated but you want them to be. For example, when the user holds the mouse down over a scroll button but doesn't move it, no events are generated after the mouse-down event. The Application Kit's scrolling mechanism then starts and uses a stream of periodic events to keep the document scrolling at a reasonable pace until the user releases the mouse. When a mouse-up event occurs, the scrolling mechanism terminates the periodic event stream.

You use the `NSEvent` class method `startPeriodicEventsAfterDelay:withPeriod:` to generate periodic events and have them placed in the event queue at a certain frequency. When you no longer need them, turn off the flow of periodic events by invoking `stopPeriodicEvents`. Unlike key and mouse events, periodic events aren't dispatched to a window object. The application must retrieve them explicitly using the `NSApplication` method `nextEventMatchingMask:untilDate:inMode:dequeue:`, typically in a modal loop. An application can have only one stream of periodic events active for each thread. You use periodic events instead of timers because `NSPeriodic` events are delivered along with other events; this makes for a responder object to look for periodic events along with mouse-up and mouse-dragged events. such as is done (for example) during scrolling.

The remaining event types—`NSAppKitDefined`, `NSSystemDefined`, and `NSApplicationDefined`—are less structured, containing only generic subtype and data fields. Of these three miscellaneous event types, only `NSApplicationDefined` is of real use to application programs. It allows the application to generate custom events and insert them into the event queue. Each such event can have a subtype and two additional codes to differentiate it from others. The `NSEvent` method `otherEventWithType:location:modifierFlags:timestamp:windowNumber:context:subtype:data1:data2:` creates one of these events, and the `subtype`, `data1`, and `data2` methods return the information specific to these events.

Event Handling Basics

Some tasks found in event-handling code are common to events of more than one type. The following sections describe these basic event-handling tasks. Some of the information presented in this chapter is discussed at length, but using different examples, in the "Creating a Custom View" chapter of *View Programming Guide*.

Preparing a Custom View for Receiving Events

Although any type of responder object can handle events, `NSView` objects are by far the most common recipient of events. They are typically the first objects to respond to both mouse events and key events, often changing their appearance in reaction to those events. Many Application Kit classes are implemented to handle events, but by itself `NSView` does not. When you create your own custom view and you want it to respond to events, you have to add event-handling capabilities to it.

The bare-minimum steps required for this are few:

- If your custom view should be first responder for key events or action messages, override `acceptsFirstResponder`.
- Implement one or more `NSResponder` methods to handle events of particular types.
- If your view is to handle action messages passed to it via the responder chain, implement the appropriate action methods.

A view that is first responder accepts key events and action messages before other objects in a window. It also typically takes part in the key loop (see "Keyboard Interface Control" (page 75)). By default, view objects refuse first-responder status by returning `NO` in `acceptsFirstResponder`. If you want your custom view to respond to key events and actions, your class must override this method to return `YES`:

```
- (BOOL)acceptsFirstResponder {
    return YES;
}
```

> **Note:** A view that becomes first responder in order to respond to key events or action messages should reflect this status by drawing a focus ring around itself. The focus ring indicates that the object is the current first responder for key events.

The second item in the list above—overriding an `NSResponder` method to handle an event—is of course a large subject and what most remaining chapters in this document are about. But there are a few basic guidelines to consider for event messages:

- If necessary, examine the passed-in `NSEvent` object to verify that it's an event you handle and, if so, find out how to handle it.

  Call the appropriate `NSEvent` methods to help make this determination. For example, you might see which modifier keys were pressed (`modifierFlags`), find out whether a mouse-down event is a double- or triple-click (`clickCount`), or, for key events, get the associated characters (`characters` and `charactersIgnoringModifiers`).

- If your implementation of an `NSResponder` method such as `mouseDown:` completely handles an event it should not invoke the superclass implementation of that method.

  `NSView` inherits the `NSResponder` implementations of the methods for handling mouse events; in these methods, `NSResponder` simply passes the message up the responder chain. One of these objects in the responder chain can implement a method in a way that ensures your custom view won't see subsequent related mouse events. Because of this, custom `NSView` objects should not invoke `super` in their implementations of `NSResponder` mouse-event-handling methods such as `mouseDown:`, `mouseDragged:` and `mouseUp:` unless it is known that the inherited implementation provides some needed functionality.

- If you do not handle the event, pass it on up the responder chain. Although you can directly forward the message to the next responder, it's generally better to forward the message to your superclass. If your superclass doesn't handle the event, it forwards the message to its superclass, and so on, until `NSResponder` is reached. By default `NSResponder` passes all event messages up the responder chain. For example, instead of this:

```
- (void)mouseDown:(NSEvent *)theEvent {
    // determine if I handle theEvent
    // if not...
    [[self nextResponder] mouseDown:theEvent];
}
```

  do this:

```
- (void)mouseDown:(NSEvent *)theEvent {
    // determine if I handle theEvent
    // if not...
    [super mouseDown:theEvent];
}
```

- If your subclass needs to handle particular events some of the time—only some typed characters, perhaps—then it must override the event method to handle the cases it's interested in and to invoke the superclass implementation otherwise. This allows a superclass to catch the cases it's interested in, and ultimately allows the event to continue on its way along the responder chain if it isn't handled.
- If you deliberately want to bypass the superclass implementation of an `NSResponder` method—say your superclass is `NSForm`—and yet pass an event up the responder chain, resend the message to `[self nextResponder]`.

Implementing Action Methods

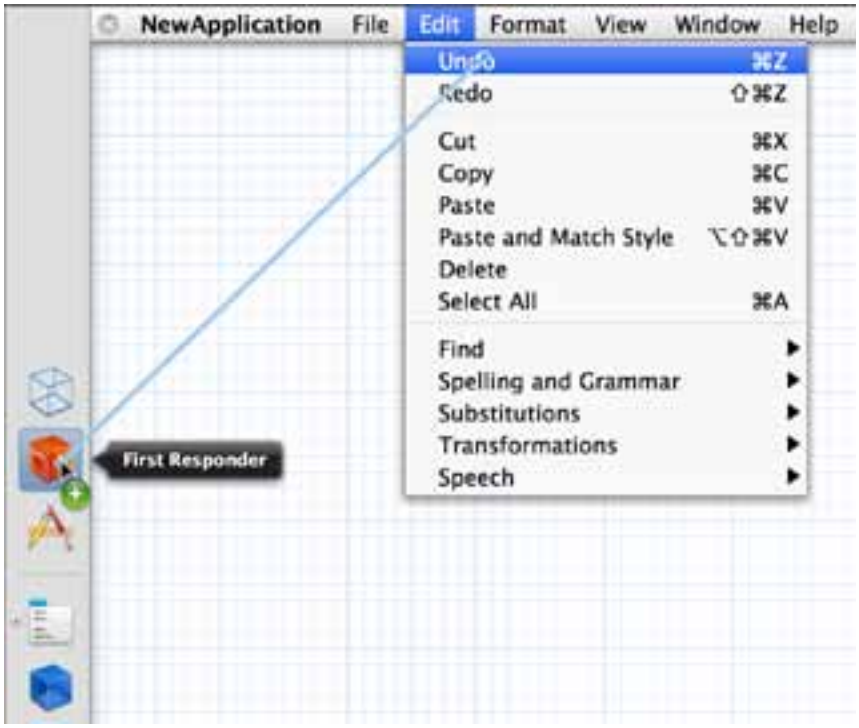
Action messages are typically sent by `NSMenuItem` objects or by `NSControl` objects. The latter objects usually work together with one or more `NSCell` objects. The cell object stores a method selector identifying the action message to be sent and a reference to the target object. (A menu item encapsulates its own action and target data.) When a menu item or control object is clicked or otherwise manipulated, it gets the action selector and target object—in the control's case, from one of its cells—and sends the message to the target.

> **Note:** For more on the target-action mechanism and action methods, see the appropriate sections in "Communicating with Objects" in *Cocoa Fundamentals Guide* . Also, for information about untargeted action messages and the path they take up the responder chain, see "Action Messages" (page 19) and "Responder Chain for Action Messages" (page 23).

You can set the action selector and target programmatically using, respectively, the methods `setAction:` and `setTarget:` (declared by `NSActionCell`, `NSMenuItem`, and other classes). However, you typically specify these in Interface Builder. In this application, you connect a control object to another object (the target) in the nib file by Control-dragging from the control object to the target and then selecting the action method of the

target to invoke. If you want the action message to be untargeted, you can either set the target to `nil` programmatically or, in Interface Builder, make a connection between the menu item or control and the First Responder icon in the nib file window, as shown in Figure 3-1.

Figure 3-1 Connecting an untargeted action in Interface Builder



From Interface Builder you can generate a header file and implementation file for your Xcode project that include a declaration and skeletal implementation, respectively, for each action method defined for a class. These Interface Builder–defined methods have a return "value" of `IBAction`, which acts as a tag to indicated that the target-action connection is archived in a nib file. You can also add the declarations and skeletal implementations of action methods yourself; in this case, the return type is `void`.) The remaining required part of the signature is a single parameter typed as `id` and named (by convention) `sender`.

Listing 3-1 illustrates a straightforward implementation of an action method that toggles a clock's AM-PM indicator when a user clicks a button.

Listing 3-1 Simple implementation of an action method

```
- (IBAction)toggleAmPm:(id)sender {
    [self incrementHour:12 andMinute: 0];
}
```

Action methods, unlike `NSResponder` event methods, don't have default implementations, so responder subclasses shouldn't blindly forward action messages to `super`. The passing of action messages up the responder chain in the Application Kit is predicated merely on whether an object responds to the method, unlike with the passing of event messages. Of course, if you know that a superclass does in fact implement the method, you can pass it on up from your subclass, but otherwise don't.

An important feature of action messages is that you can send messages back to `sender` to get further information or associated data. For example, the menu items in a given menu might have represented objects assigned to them; for example, a menu item with a title of "Red" might have a represented object that is an `NSColor` object. You can access this object by sending `representedObject` to `sender`.

You can take the messaging-back feature of action methods one step further by using it to dynamically change `sender`'s target, action, title, and similar attributes. Here is a simple test case: You want to control a progress indicator (an `NSProgressIndicator` object) with a single button; one click of the button starts the indicator and changes the button's title to "Stop", and then the next click stops the indicator and changes the title to back to "Start". Listing 3-2 shows one way to do this.

Listing 3-2 Resetting target and action of `sender`—good implementation

```
- (IBAction)controlIndicator:(id)sender
{
    [[sender cell] setTarget:indicator]; // indicator is NSProgressIndicator
    if ( [[sender title] isEqualToString:@"Start"] ) {
        [[sender cell] setAction:@selector(startAnimation:)];
        [sender setTitle:@"Stop"];
    } else {
        [[sender cell] setAction:@selector(stopAnimation:)];
        [sender setTitle:@"Start"];
    }
    [[sender cell] performClick:self];
    // set target and action back to what they were
    [[sender cell] setAction:@selector(controlIndicator:)];
    [[sender cell] setTarget:self];
}
```

However, this implementation requires that the target and action information be set back to what they were after the redirected action message is sent via `performClick:`. You could simplify this implementation by invoking directly `sendAction:to:from:`, the method used by the application object (`NSApp`) to dispatch action messages (see Listing 3-3).

Listing 3-3 Resetting target and action of `sender`—better implementation

```
- (IBAction)controlIndicator:(id)sender
{
    SEL theSelector;
    if ( [[sender title] isEqualToString:@"Start"] ) {
        theSelector = @selector(startAnimation:);
        [sender setTitle:@"Stop"];
    } else {
        theSelector = @selector(stopAnimation:);
        [sender setTitle:@"Start"];
    }
    [NSApp sendAction:theSelector to:indicator from:sender];
}
```

In keyboard action messages, an action method is invoked as a result of a particular key-press being interpreted through the key bindings mechanism. Because such messages are so closely connected to specific key events, the implementation of the action method can get the event by sending `currentEvent` to `NSApp` and then query the `NSEvent` object for details. Listing 3-7 (page 46) gives an example of this technique. See "The Path of Key Events" (page 16) for a summary of keyboard action messages; also see "Key Bindings" (page 105) for a description of that mechanism.

Getting the Location of an Event

You can get the location of a mouse or tablet-pointer event by sending `locationInWindow` to an `NSEvent` object. But, as the name of the method denotes, this location (an `NSPoint` structure) is in the base coordinate system of a window, not in the coordinate system of the view that typically handles the event. Therefore a view must convert the point to its own coordinate system using the method `convertPoint:fromView:`, as shown in Listing 3-4.

Listing 3-4 Converting a mouse-dragged location to be in a view's coordinate system

```
- (void)mouseDragged:(NSEvent *)event {
    NSPoint eventLocation = [event locationInWindow];
    center = [self convertPoint:eventLocation fromView:nil];
    [self setNeedsDisplay:YES];
}
```

The second parameter of `convertPoint:fromView:` is `nil` to indicate that the conversion is from the window's base coordinate system.

Keep in mind that the `locationInWindow` method is not appropriate for key events, periodic events, or any other type of event other than mouse and tablet-pointer events.

Testing for Event Type and Modifier Flags

On occasion you might need to discover the type of an event. However, you do not need to do this within an event-handling method of `NSResponder` because the type of the event is apparent from the method name: `rightMouseDragged:`, `keyDown:`, `tabletProximity:`, and so on. But from elsewhere in an application you can always obtain the currently handled event by sending `currentEvent` to `NSApp`. To find out what type of event this is, send `type` to the `NSEvent` object and then compare the returned value with one of the `NSEventType` constants. Listing 3-5 gives an example of this.

Listing 3-5 Testing for event type

```
NSEvent *currentEvent = [NSApp currentEvent];
NSPoint mousePoint = [controlView convertPoint: [currentEvent locationInWindow]
fromView:nil];
switch ([currentEvent type]) {
    case NSLeftMouseDown:
    case NSLeftMouseDragged:
        [self doSomethingWithEvent:currentEvent];
        break;
    default:
        // If we find anything other than a mouse down or dragged we are done.
         return YES;
}
```

A common test performed in event-handling methods is finding out whether specific modifier keys were pressed at the same moment as a key-press, mouse click, or similar user action. The modifier key usually imparts special significance to the event. The code example in Listing 3-6 shows an implementation of `mouseDown:` that determines whether the Command key was pressed while the mouse was clicked. If it was, it rotates the receiver (a view) 90 degrees. The identification of modifier keys requires the event handler to send `modifierFlags` to the passed-in event object and then perform a bitwise-AND operation on the returned value using one or more of the modifier mask constants declared in `NSEvent.h`.

Listing 3-6 Testing for modifier keys pressed—event method

```
- (void)mouseDown:(NSEvent *)theEvent {


    //  if Command-click rotate 90 degrees
    if ([theEvent modifierFlags] & NSCommandKeyMask) {
        [self setFrameRotation:[self frameRotation]+90.0];
        [self setNeedsDisplay:YES];
    }
}
```

You can test an event object to find out if any from a set of modifier keys was pressed by performing a bitwise-OR with the modifier-mask constants, as in Listing 3-7. (Note also that this example shows the use of the `currentEvent` method in a keyboard action method.)

Listing 3-7 Testing for modifier keys pressed—action method

```
- (void)moveLeft:(id)sender {
    // Use left arrow to decrement the time.  If a shift key is down, use a big
step size.
    BOOL shiftKeyDown = ([[NSApp currentEvent] modifierFlags] &
        (NSShiftKeyMask | NSAlphaShiftKeyMask)) !=0;
    [self incrementHour:0 andMinute:-(shiftKeyDown ? 15 : 1)];
}
```

Further, you can look for certain modifier-key combinations by linking the individual modifier-key tests together with the logical AND operator (&&). For example, if the example method in Listing 3-6 (page 46) were to look for Command-Shift-click rather than Command-click, the complete test would be the following:

```
    if ( ( [theEvent modifierFlags] & NSCommandKeyMask ) &&
        ( [theEvent modifierFlags] & NSShiftKeyMask ) )
```

In addition to testing an `NSEvent` object for individual event types, you can also restrict the events fetched from the event queue to specified types. You can perform this filtering of event types in the `nextEventMatchingMask:untilDate:inMode:dequeue:` method (`NSApplication` and `NSWindow`) or the `nextEventMatchingMask:` method of `NSWindow`. The second parameter of all these methods takes one

or more of the event-type mask constants declared in `NSEvent.h`—for example `NSLeftMouseDraggedMask`, `NSFlagsChangedMask`, and `NSTabletProximityMask`. You can either specify these constants individually or perform a bitwise-OR on them.

Because the `nextEventMatchingMask:untilDate:inMode:dequeue:` method is used almost exclusively in closed loops for processing a related series of mouse events, the use of it is described in "Handling Mouse Events" (page 51).

Responder-Related Tasks

The following sections describe tasks related to the first-responder status of objects.

Determining First-Responder Status

Usually an `NSResponder` object can always determine if it's currently the first responder by asking its window (or itself, if it's an `NSWindow` object) for the first responder and then comparing itself to that object. You ask an `NSWindow` object for the first responder by sending it a `firstResponder` message. For an `NSView` object, this comparison would look like the following bit of code:

```
if ([[self window] firstResponder] == self) {
      // do something based upon first-responder status
}
```

A complication of this simple scenario occurs with text fields. When a text field has input focus, it is not the first responder. Instead, the field editor for the window is the first responder; if you send `firstResponder` to the `NSWindow` object, an `NSTextView` object (the field editor) is what is returned. To determine if a given `NSTextField` is currently active, retrieve the first responder from the window and find out it is an `NSTextView` object and if its delegate is equal to the `NSTextField` object. Listing 3-8 shows how you might do this.

Listing 3-8 Determining if a text field is first responder

```
if ( [[[self window] firstResponder] isKindOfClass:[NSTextView class]] &&
   [window fieldEditor:NO forObject:nil] != nil ) {
        NSTextField *field = [[[self window] firstResponder] delegate];
        if (field == self) {
            // do something based upon first-responder status
        }
}
```

The control that a field editor is editing is always the current delegate of the field editor, so (as the example shows) you can obtain the text field by asking for the field editor's delegate. For more on the field editor, see "Text Fields, Text Views, and the Field Editor".

Setting the First Responder

You can programmatically change the first responder by sending `makeFirstResponder:` to an `NSWindow` object; the argument of this message must be a responder object (that is, an object that inherits from `NSResponder`). This message initiates a kind of protocol in which one object loses its first responder status and another gains it.

1. `makeFirstResponder:` always asks the current first responder if it is ready to resign its status by sending it `resignFirstResponder`.
2. If the current first responder returns `NO` when sent this message, `makeFirstResponder:` fails and likewise returns `NO`.

   A view object or other responder may decline to resign first responder status for many reasons, such as when an action is incomplete.
3. If the current first responder returns `YES` to `resignFirstResponder`, then the new first responder is sent a `becomeFirstResponder` message to inform it that it can be the first responder.
4. This object can return `NO` to reject the assignment, in which case the `NSWindow` itself becomes the first responder.

Figure 3-2 and Figure 3-3 illustrate two possible outcomes of this protocol.

Figure 3-2 Making a view a first responder—current view refuses to resign status

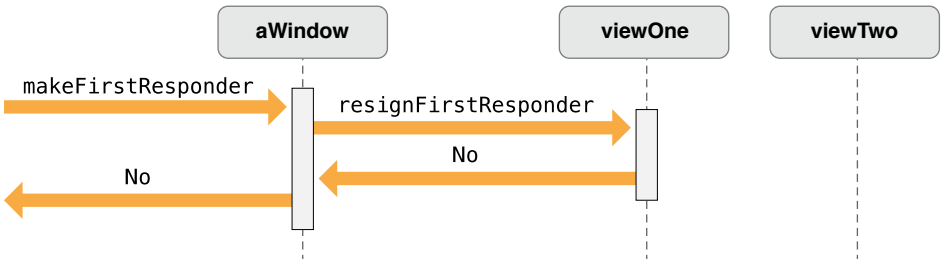
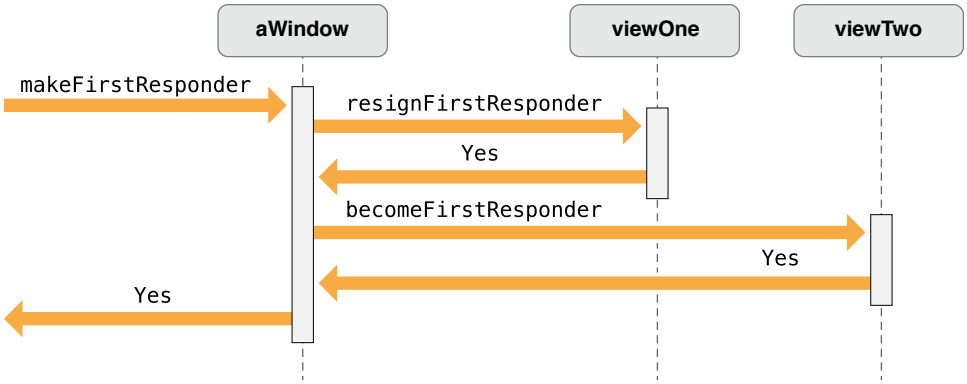


Figure 3-3 Making a view a first responder—new view becomes first responder



Listing 3-9 shows a custom `NSCell` class (in this case a subclass of `NSActionCell`) implementing `resignFirstResponder` and `becomeFirstResponder` to manipulate the keyboard focus ring of its superclass.

Listing 3-9 Resigning and becoming first responder

```
- (BOOL)becomeFirstResponder {
    BOOL okToChange = [super becomeFirstResponder];
    if (okToChange) [self setKeyboardFocusRingNeedsDisplayInRect: [self bounds]];
    return okToChange;
}

- (BOOL)resignFirstResponder {
    BOOL okToChange = [super resignFirstResponder];
    if (okToChange) [self setKeyboardFocusRingNeedsDisplayInRect: [self bounds]];
    return okToChange;
}
```

You can also set the *initial* first responder for a window—that is, the first responder set when the window is first placed onscreen. You can set the initial first responder programmatically by sending `setInitialFirstResponder:` to an `NSWindow` object. You can also set it when you create a user interface in Interface Builder. To do this, complete the following steps.

1. Control-drag from the window icon in the nib file window to an `NSView` object in the user interface.
2. In the Connections pane of the inspector, select the `initialFirstResponder` outlet and click Connect.

After `awakeFromNib` is called for all objects in a nib file, `NSWindow` makes the first responder whatever was set as initial first responder in the nib file. Note that you should not send `setInitialFirstResponder:` to an `NSWindow` object in `awakeFromNib` and expect the message to be effective.

Handling Mouse Events

Mouse events are one of the two most frequent kinds of events handled by an application (the other kind being, of course, key events). Mouse clicks—which involve a user pressing and then releasing a mouse button—generally indicate selection, but what the selection means is left up to the object responding to the event. For example, a mouse click could tell the responding object to alter its appearance and then send an action message. Mouse drags generally indicate that the receiving view should move itself or a drawn object within its bounds. The following sections describe how you might handle mouse-down, mouse-up, and mouse-drag events.

> **Note:** This chapter discusses those mouse events that an `NSWindow` object delivers to its views through the event-dispatch mechanism: mouse-down, mouse-up, mouse-dragged, and (to a lesser degree) mouse-moved events. It does not describe how to handle mouse-entered and mouse-exited events, which (possibly along with mouse-moved events) are used in mouse tracking. See "Using Tracking-Area Objects" (page 78) for information on this subject.

Overview of Mouse Events

Before going into the "how to" of mouse-event handling, let's review some central facts about mouse events discussed in "Event Architecture" (page 10), "Event Objects and Types" (page 27), and "Event Handling Basics" (page 39):

- Mouse events are dispatched by an `NSWindow` object to the `NSView` object over which the event occurred.
- Subsequent key events are dispatched to that view object if it accepts first responder status.
- If the user clicks a view that isn't in the key window, by default the window is brought forward and made key, but the mouse event is not dispatched.
- Mouse events are of various event types related to the mouse button pressed (left, right, other) and the nature of the action on the mouse button. Each event type is, in turn, related to an `NSResponder` method, as Table 4-1 shows for left-button mouse events.

Table 4-1 Type constants and methods related to left-button mouse events

Action	Event type (left mouse button)	Mouse-event method invoked (left mouse button)
Press down the button	`NSLeftMouseDown`	`mouseDown:`

Action	Event type (left mouse button)	Mouse-event method invoked (left mouse button)
Move the mouse while pressing the button	`NSLeftMouseDragged`	`mouseDragged:`
Release the button	`NSLeftMouseUp`	`mouseUp:`
Move the mouse without pressing any button	`NSMouseMoved`	`mouseMoved:`

> **Note:** Because mouse-moved events occur so frequently that they can quickly flood the event-dispatch machinery, an `NSWindow` object by default does not receive them from the global `NSApplication` object. However, you can specifically request these events by sending the `NSWindow` object an `setAcceptsMouseMovedEvents:` message with an argument of `YES`.

Right-mouse events are defined by the Application Kit to open contextual menus, but you can override this behavior if necessary (in, for example, `rightMouseDown:`). The Application Kit does not define any default behavior for third ("other") mouse buttons.

- The general sequence of mouse events is: mouse-down, mouse-dragged (multiple), mouse-up.

  If dispatch of them is turned on, mouse-moved events may only occur between a mouse-up and the next mouse-down. They do not occur between a mouse-down and the subsequent mouse-up.

- Mouse events (as `NSEvent` objects) can have subtypes. Currently, these subtypes are restricted to tablet events (particularly tablet-pointer events) but more subtypes could be added in the future. (See "Tablet Events" (page 33) for more information.)

Handling Mouse Clicks

One of the earliest things to consider in handling mouse-down events is whether the receiving `NSView` object should become the first responder, which means that it will be the first candidate for subsequent key events and action messages. Views that handle graphic elements that the user can select—drawing shapes or text, for example—should typically accept first responder status on a mouse-down event by overriding the `acceptsFirstResponder` method to return `YES`, as discussed in "Preparing a Custom View for Receiving Events" (page 39).

By default, a mouse-down event in a window that isn't the key window simply brings the window forward and makes it key; the event isn't sent to the `NSView` object over which the mouse click occurs. The `NSView` can claim an initial mouse-down event, however, by overriding `acceptsFirstMouse:` to return `YES`. The argument of this method is the mouse-down event that occurred in the non-key window, which the view object can examine to determine whether it wants to receive the mouse event and potentially become first responder. You want the default behavior of this method in, for example, a control that affects the selected object in a window. However, in certain cases it's appropriate to override this behavior, such as for controls that should receive `mouseDown:` messages even when the window is inactive. Examples of controls that support this click-through behavior are the title-bar buttons of a window.

In your implementation of an `NSResponder` mouse-event method, often the first thing you might do is examine the passed-in `NSEvent` object to decide *if* this is an event you want to handle. If it is an event that you handle, then you may need to extract information from the `NSEvent` object to help you handle it. Specifically, you can get the following information from the `NSEvent` object:

- Get the location of the mouse event in base coordinates (`locationInWindow`) and then convert this location to the receiving view's coordinate system; see "Getting the Location of an Event" (page 44) for details.
- See if any modifier keys were pressed when the mouse button was clicked (`modifierFlags`); this procedure is described in "Testing for Event Type and Modifier Flags" (page 45). An application may define modifier keys to change the significance of a mouse event.
- Find out how many mouse clicks occurred in quick succession (`clickCount`); multiple mouse clicks are conceptually treated as a single mouse-down event within a narrow time threshold (although they arrive in a series of `mouseDown:` messages). As with modifier keys, a double- or triple-click can change the significance of a mouse event for an application. (See Listing 4-3 (page 55) for an example.)
- If the interval between mouse clicks is important, you can send `timestamp` to the `NSEvent` object and record the moment each event occurred.
- If the change in position of the mouse between subsequent events is important, you can find out the delta values for the x-coordinate and y-coordinate using, respectively, `deltaX` and `deltaY`.

Many view objects in the Application Kit (such as controls and menu items) change their appearance in response to mouse-down events, sometimes only until the subsequent mouse-up event. Doing this provides visual confirmation to the user that their action is effective or that the clicked object is now selected. Listing 4-1 shows a simple example of this.

Listing 4-1 Simple handling of mouse click—changing view's appearance

```
- (void)mouseDown:(NSEvent *)theEvent {
    [self setFrameColor:[NSColor redColor]];
```

```
    [self setNeedsDisplay:YES];
}

- (void)mouseUp:(NSEvent *)theEvent {
    [self setFrameColor:[NSColor greenColor]];
    [self setNeedsDisplay:YES];
}

- (void)drawRect:(NSRect)rect {
    [[self frameColor] set];
    NSRectFill(rect);
}
```

But many view objects, particularly controls and cells, do more than simply change their appearance in response to mouse clicks. One common paradigm is for the clicked view to send an action message to a target object (where both action and target are settable properties of the view). As shown in Listing 4-2, the view typically sends the message on `mouseUp:` rather than `mouseDown:`, thus giving users an opportunity to change their minds mid-click.

Listing 4-2 Simple handling of mouse click—sending an action message

```
- (void)mouseDown:(NSEvent *)theEvent {
    [self setFrameColor:[NSColor redColor]];
    [self setNeedsDisplay:YES];
}

- (void)mouseUp:(NSEvent *)theEvent {
    [self setFrameColor:[NSColor greenColor]];
    [self setNeedsDisplay:YES];
    [NSApp sendAction:[self action] to:[self target] from:self];
}

- (SEL)action {return action; }

- (void)setAction:(SEL)newAction {
```

```
    action = newAction;
}

- (id)target { return target; }

- (void)setTarget:(id)newTarget {
    target = newTarget;
}
```

Listing 4-3 gives a more complex, real-world example. (It's from the example project for the Sketch application.) This implementation of `mouseDown:` determines if users double-clicked a graphical object and, if they did, enables the editing of that object. Otherwise, if a palette object is selected, it creates an instance of that object at the location of the mouse click.

Listing 4-3 Handling a mouse-down event—Sketch application

```
- (void)mouseDown:(NSEvent *)theEvent {
    Class theClass = [[SKTToolPaletteController sharedToolPaletteController]
currentGraphicClass];
    if ([self editingGraphic]) {
        [self endEditing];
    }
    if ([theEvent clickCount] > 1) {
        NSPoint curPoint = [self convertPoint:[theEvent locationInWindow]
fromView:nil];
        SKTGraphic *graphic = [self graphicUnderPoint:curPoint];
        if (graphic && [graphic isEditable]) {
            [self startEditingGraphic:graphic withEvent:theEvent];
            return;
        }
    }
    if (theClass) {
        [self clearSelection];
        [self createGraphicOfClass:theClass withEvent:theEvent];
    } else {
        [self selectAndTrackMouseWithEvent:theEvent];
    }
```

```
}
```

The classes of the Application Kit that implement controls manage this target-action behavior for you.

Handling Mouse Dragging Operations

Mouse-down and mouse-up events, which are the events associated with mouse clicks, are not the only types of mouse events dispatched in an application. Views can also receive mouse-dragged events when a user moves a mouse while pressing down a mouse button. A view typically interprets mouse-dragged events as commands to move itself by altering its frame location or to move a region within its bounds by redrawing it. However, other interpretations of mouse-dragged events are possible; for example, a view could respond to mouse-dragged events by magnifying the region that the mouse pointer is dragged over.

With the Application Kit you can take one of two general approaches when handling mouse-dragged events. The first approach is overriding the three `NSResponder` methods `mouseDown:`, `mouseDragged:`, and `mouseUp:` (for left mouse-button operations). For each dragging sequence, the Application Kit sends a `mouseDown:` message to a responder object, then sends one or more `mouseDragged:` messages, and ends the sequence with a `mouseUp:` message. "The Three-Method Approach" (page 57) describes this approach.

The other approach treats the mouse events in a dragging sequence as a single event, from mouse down, through dragging, to mouse up. To do this, the responder object must usually short-circuit the application's normal event loop by entering a event-tracking loop to filter and process only mouse events of interest. For example, an `NSButton` object highlights itself upon a mouse-down event, then follows the mouse location during dragging, highlighting when the mouse is inside and unhighlighting when the mouse is outside. If the mouse is inside on the mouse-up event, the button object sends its action message. This approach is described in "The Mouse-Tracking Loop Approach" (page 59).

Both of these approaches have their advantages and drawbacks. Establishing a mouse-tracking loop gives you greater control over the way other events interact with your application during a dragging operation. However, the application's main thread is unable to process any other requests during an event-tracking loop and timers might not fire as expected. The mouse-tracking approach is more efficient because it typically requires less code and allows all dragging variables to be local. However, the class implementing it becomes more difficult to extend without the subclass reimplementing all the dragging code.

Implementing the individual `mouseDown:`, `mouseDragged:`, and `mouseUp:` methods is often a better design choice when writing an event-driven application. Each of the methods have a clearly defined scope, which often leads to clearer code. This approach also makes it much easier for subclasses to override behavior for handling mouse-down, mouse-dragged, and mouse-up events. However, this technique can require more code and instance variables.

The Three-Method Approach

To handle a mouse-dragging operation, you can override the three `NSResponder` methods that mark the discrete stages of a mouse-dragging operation: `mouseDown:`, `mouseDragged:`, and `mouseUp:` (or, for right-mouse dragging, `rightMouseDown:`, `rightMouseDragged:`, and `rightMouseUp:`). A mouse-dragging sequence consists of one `mouseDown:` message, followed by (typically) multiple `mouseDragged:` messages, and a concluding `mouseUp:` message.

The subclass implementing these methods often has to declare instance variables to hold the changing values or states of various things between successive events. These things could be geometric entities such as rectangles or points (corresponding to view frames or regions within views) or they could be simple Boolean values indicating, for example, that an object is selected. In the `mouseDown:` method, the view generally initializes any dragging-related instance variables; in the `mouseDragged:` method it might update those instance variables or check them prior to performing an action; and in the `mouseUp:` method, it often resets those instance variables to their initial values.

Because mouse-dragging operations often redraw an object in the incrementally changing locations where users drag that object, implementations of the three mouse-dragging methods usually need to find the location of each mouse event in the view's coordinate system. As explained in "Getting the Location of an Event" (page 44), this requires the view to send `locationInWindow` to the passed-in `NSEvent` object and then use the `convertPoint:fromView:` method to convert the resulting location to the local coordinate system. When changes in location or geometry require the view to redraw itself or a portion of itself, the view marks the areas needing display using the `setNeedsDisplay:` or `setNeedsDisplayInRect:` method and the view is later asked to redraw itself (in `drawRect:`) through the auto-display mechanism (assuming that mechanism is not turned off).

Listing 4-4 illustrates the use of dragging-related instance variables, getting the mouse location in local coordinates (and testing whether that location is in a specific region), and marking that region of the receiving view for redisplay.

Listing 4-4 Handling a mouse-dragging operation—three-method approach

```
- (void)mouseDown:(NSEvent *)theEvent {

    // mouseInCloseBox and trackingCloseBoxHit are instance variables

    if (mouseInCloseBox = NSPointInRect([self convertPoint:[theEvent
locationInWindow] fromView:nil], closeBox)) {

        trackingCloseBoxHit = YES;

        [self setNeedsDisplayInRect:closeBox];

    }

    else if ([theEvent clickCount] > 1) {

        [[self window] miniaturize:self];
```

```
        return;
    }
}

- (void)mouseDragged:(NSEvent *)theEvent {
    NSPoint windowOrigin;
    NSWindow *window = [self window];

    if (trackingCloseBoxHit) {
        mouseInCloseBox = NSPointInRect([self convertPoint:[theEvent
locationInWindow] fromView:nil], closeBox);
        [self setNeedsDisplayInRect:closeBox];
        return;
    }

    windowOrigin = [window frame].origin;

    [window setFrameOrigin:NSMakePoint(windowOrigin.x + [theEvent deltaX],
windowOrigin.y - [theEvent deltaY])];
}

- (void)mouseUp:(NSEvent *)theEvent {
   if (NSPointInRect([self convertPoint:[theEvent locationInWindow] fromView:nil],
 closeBox)) {
        [self tryToCloseWindow];
        return;
    }
    trackingCloseBoxHit = NO;
    [self setNeedsDisplayInRect:closeBox];
}
```

The Mouse-Tracking Loop Approach

The mouse-tracking technique for handling mouse-dragging operations is applied in a single method, usually (but not necessarily) in `mouseDown:`. The implementing responder object first declares and possibly initializes one or more local variables to use within the loop. Often one of these variables holds a value, often Boolean, that is used to control the loop. When some condition is met, typically the receipt of a mouse-up event, the variable value is changed; when this variable is tested next time through the loop, control exits the loop.

The central method of a mouse-tracking loop is the `NSApplication` method `nextEventMatchingMask:untilDate:inMode:dequeue:` and the `NSWindow` method of the same name. These methods fetch events from the event queue that are of the types specified by one or more type-mask constants; for mouse-dragging, these constants are typically `NSLeftMouseDraggedMask` and `NSLeftMouseUpMask`. Events of other types are left in the queue. (The run-loop mode parameter of both `nextEventMatchingMask:untilDate:inMode:dequeue:` methods should be `NSEventTrackingRunLoopMode`.)

After receiving a mouse-down event, fetch subsequent mouse events within the loop using `nextEventMatchingMask:untilDate:inMode:dequeue:`. Process `NSLeftMouseDragged` events as you would process them in the `mouseDragged:` method (described in "The Three-Method Approach" (page 57)); similarly, handle `NSLeftMouseUp` events as you would handle them in `mouseUp:`. Usually mouse-up events indicate that execution control should break out of the loop.

The `mouseDown:` method template in Listing 4-5 shows one possible kind of modal event loop.

Listing 4-5 Handling mouse-dragging in a mouse-tracking loop—simple example

```
- (void)mouseDown:(NSEvent *)theEvent {
    BOOL keepOn = YES;
    BOOL isInside = YES;
    NSPoint mouseLoc;

    while (keepOn) {
        theEvent = [[self window] nextEventMatchingMask: NSLeftMouseUpMask |
                NSLeftMouseDraggedMask];
        mouseLoc = [self convertPoint:[theEvent locationInWindow] fromView:nil];
        isInside = [self mouse:mouseLoc inRect:[self bounds]];

        switch ([theEvent type]) {
            case NSLeftMouseDragged:
                    [self highlight:isInside];
```

```
                break;
            case NSLeftMouseUp:
                if (isInside) [self doSomethingSignificant];
                [self highlight:NO];
                keepOn = NO;
                break;
            default:
                /* Ignore any other kind of event. */
                break;
        }

    };

    return;
}
```

This loop converts the mouse location and checks whether it's inside the receiver. It highlights itself using the fictional `highlight:` method and, on receiving a mouse-up event, it invokes `doSomethingSignificant` to perform an important action. Instead of merely highlighting, a custom `NSView` object might move a selected object, draw a graphic image according to the mouse's location, and so on.

Listing 4-6 is a slightly more complicated example that includes the use of an autorelease pool and testing for a modifier key.

Listing 4-6 Handling mouse-dragging in a mouse-tracking loop—complex example

```
- (void)mouseDown:(NSEvent *)theEvent
{
    if ([theEvent modifierFlags] & NSAlternateKeyMask)  {
        BOOL                dragActive = YES;
        NSPoint             location = [renderView convertPoint:[theEvent
locationInWindow] fromView:nil];
        NSAutoreleasePool   *myPool = nil;
        NSEvent*            event = NULL;
        NSWindow            *targetWindow = [renderView window];
```

```
        myPool = [[NSAutoreleasePool alloc] init];
        while (dragActive) {
            event = [targetWindow nextEventMatchingMask:(NSLeftMouseDraggedMask |
 NSLeftMouseUpMask)
                                    untilDate:[NSDate distantFuture]
                                    inMode:NSEventTrackingRunLoopMode
                                    dequeue:YES];
            if(!event)
                continue;
            location = [renderView convertPoint:[event locationInWindow]
fromView:nil];
            switch ([event type]) {
                case NSLeftMouseDragged:
                    annotationPeel = (location.x * 2.0 / [renderView
bounds].size.width);
                    [imageLayer showLens:(annotationPeel <= 0.0)];
                  [peelOffFilter setValue:[NSNumber numberWithFloat:annotationPeel]
 forKey:@"inputTime"];
                    [self refresh];
                    break;

                case NSLeftMouseUp:
                    dragActive = NO;
                    break;

                default:
                    break;
            }
        }
        [myPool release];
    } else {
        // other tasks handled here......
    }
}
```

A mouse-tracking loop is driven only as long as the user actually moves the mouse. It won't work, for example, to cause continual scrolling if the user presses the mouse button but never moves the mouse itself. For this, your loop should start a periodic event stream using the `NSEvent` class method `startPeriodicEventsAfterDelay:withPeriod:`, and add `NSPeriodicMask` to the mask bit-field passed to `nextEventMatchingMask:`. In the `switch` statement the implementing view object can then check for events of type `NSPeriodic` and take whatever action it needs to—scrolling a document view or moving a step in an animation, for example. If you need to check the mouse location during a periodic event, you can use the `NSWindow` method `mouseLocationOutsideOfEventStream`.

Filtering Out Key Events During Mouse-Tracking Operations

A potential problem with mouse-tracking code is the user pressing a key combination that is a command, such as Command-z (undo), while a tracking operation is underway. Because your mouse-tracking code (either in the three-method approach or a mouse-tracking loop) isn't looking for that key event, the code might not know how to handle that key command or could handle it wrongly, with unwelcome consequences.

The problem here with a mouse-tracking loop might not be readily apparent because, after all, the `nextEventMatchingMask:untilDate:inMode:dequeue:` loop ensures only mouse-tracking events are delivered. Why would key events be a problem? Consider the code in Listing 4-7. While this mouse-tracking loop is active, let's say the user issues some key-equivalent commands.

Listing 4-7 Typical mouse-tracking loop

```
- (void)mouseDown:(NSEvent *)theEvent {
  NSPoint pos;

  while ((theEvent = [[self window] nextEventMatchingMask:
    NSLeftMouseUpMask | NSLeftMouseDraggedMask])) {

    NSPoint pos = [self convertPoint:[theEvent locationInWindow]
                            fromView:nil];

    if ([theEvent type] == NSLeftMouseUp)
      break;

    // Do some other processing...
  }
}
```

What happens after the mouse-tracking loop concludes? After the user lets go of the mouse button, the application handles all of the pending commands corresponding to the mnemonics the user pressed during the loop. The effects of this delayed handling are probably undesirable. You can guard against this by filtering the event stream for key events and then explicitly ignoring them. Listing 4-8 shows how you can modify the code above to accomplish this (new code in italics).

Listing 4-8 Typical mouse-tracking loop—with key events negated

```
- (void)mouseDown:(NSEvent *)theEvent {
  NSPoint pos;

  while ((theEvent = [[self window] nextEventMatchingMask:
      NSLeftMouseUpMask | NSLeftMouseDraggedMask | NSKeyDownMask])) {

    NSPoint pos = [self convertPoint:[theEvent locationInWindow]
                            fromView:nil];

    if ([theEvent type] == NSLeftMouseUp)
        break;
          else if ([theEvent type] == NSKeyDown) {
                NSBeep();
                continue;
    }

    // Do some other processing...
  }
}
```

For dragging operations handled with the three-method approach, the situation with simultaneous mouse and key events is a bit different. In this case, the AppKit processes keyboard events as it normally does during tracking. If the user presses a command mnemonic, even while a tracking operation is going on, the application object dispatches the corresponding messages to its targets. So if, for example, in a drawing application the user drags a blue circle they just created and then (perhaps accidentally) presses Command-x (cut) while the mouse button is still down, the code handling the cut operation is going to run, deleting the object the user was dragging.

The solution to this problem involves a few steps:

- Declare a Boolean instance variable that reflects when a dragging operation is underway.
- Set this variable to `YES` in `mouseDown:` and reset it to `NO` in `mouseUp:`.
- Override `performKeyEquivalent:` to check the value of the instance variable and, if a dragging operation is occurring, to discard the key event.

Listing 4-9 shows the implementation code for this (`isBeingManipulated` is the Boolean instance variable).

Listing 4-9 Discarding key events during a dragging operation with the three-method approach

```
@implementation MyView

- (BOOL)performKeyEquivalent:(NSEvent *)anEvent
{
  if (isBeingManipulated) {
    if ([anEvent type] == NSKeyDown) // Can get NSKeyUp here too
      NSBeep ();
    return YES; // Claim we handled it
  }

  return NO;
}

- (void)mouseDown:(NSEvent *)anEvent
{
  isBeingManipulated = YES;
  // other code goes  here...
}

- (void)mouseUp:(NSEvent *)anEvent
{
  isBeingManipulated = NO;
  // other code goes here ...
}

@end
```

This solution to the problem is simplified and is intended for general illustration. In a real application you might want to check the event type, conditionally set the `isBeingManipulated` variable, and selectively handle key equivalents.

Handling Key Events

An OS X system generates key events when a user presses a key on a keyboard or presses several keys simultaneously. When more than one key is pressed, one or more of those keys modifies the significance of the "main" key that is pressed. The most commonly used modifier keys are the Command, Control, Option (Alt), and Shift keys. In certain contexts and combinations, key presses represent commands to the operating system or the frontmost application and not characters to be inserted into text.

This chapter discusses how you handle key events, particularly key-down events.

Overview of Key Events

The salient facts about key events, as presented in "Key Events" (page 32) and "The Path of Key Events" (page 16), are the following:

- Key events are of three specific types (`NSEventType`), each of which is associated with an `NSResponder` method:

Event-type constant	Event method
`NSKeyDown`	`keyDown:`
`NSKeyUp`	`keyUp:`
`NSFlagsChanged`	`flagsChanged:`

  The `flagsChanged:` method can be useful for detecting the pressing of modifier keys without any other key being pressed simultaneously. For example, if the user presses the Option key by itself, your responder object can detect this in its implementation of `flagsChanged:`.

- *Most* key events—that is, those representing characters to be inserted as text—are dispatched by the `NSWindow` object associated with the key window to the first responder.
- If the first responder does not handle the event, it passes the event up the responder chain (see "The Responder Chain" (page 23)).
- The delivery path of a key event varies according to whether the event represents a character, a key equivalent, a keyboard action, or a keyboard interface control command. The global application object (`NSApp`) first looks for key equivalents and then keyboard interface control commands and handles them

specially (see "The Path of Key Events" (page 16) for details). If the event is neither of these, it dispatches it to the `NSWindow` object representing the key window, which in turn dispatches the event to the first responder in a `keyDown:` message.

- The responder object determines what the key event represents and handles it appropriately. At this point, the key event could represent any one of the following things:
  - A keyboard action, which is a key or key combination bound to an action-message selector in a key bindings dictionary (see "Key Bindings" (page 105)).
  - An application-specific command or action (one not using the key bindings dictionary)
  - A character or characters to insert into text

  See "Overriding the keyDown: Method" for more information.
- As with key event messages, a keyboard action message is passed up the responder chain if the first responder does not handle it.

Overriding the keyDown: Method

Most responder objects (such as custom views) handle key events by overriding the `keyDown:` method declared by `NSResponder`. The object can handle the event in any way it sees fit. A text object typically interprets the message as a request to insert text, while a drawing object might only be interested in a few keys, such as Delete and the arrow keys as commands to delete and move selected items. As with mouse events, a responder object often wants to query the passed-in `NSEvent` object to find out more about the event and obtain the data it needs to handle it. Some of the more useful `NSEvent` methods for key events are the following:

- `characters` and `charactersIgnoringModifiers`—The responder can extract the Unicode character data associated with the event and insert it as text or interpret it as commands. The `charactersIgnoringModifiers` method ignores any modifier keystroke (except for Shift) when returning the character data. Note that both method names are plural because a keystroke can produce more than one character (for example, "à" is composed of 'a' and '`').
- `modifierFlags`—Using this method the responder can determine if any modifier keys were pressed.
- `isARepeat`—This method tells the responder whether the same key was pressed rapidly in succession.

Within an implementation of a `keyDown:` method, a responder can extract the character data contained by the associated `NSEvent` object and insert it into displayed text; or it can interpret the character data as a key or key combination that is either bound to a keyboard action or requests some application-specific behavior. However, the Application Kit provides some convenient shortcuts for doing this, described below.

Handling Keyboard Actions and Inserting Text

Responder objects that deal with text, such as text views, have to be prepared to handle key events that can either be characters to insert or keyboard actions. As noted in "The Path of Key Events" (page 16), keyboard actions are a special kind of action message that depends on the key bindings mechanism, which binds specific keystrokes (for example, Control-e) to specific commands related to the text (for example, move the insertion point to the end of the current line). These commands are implemented in methods defined by `NSResponder` to give per-view functional interpretations of those physical keystrokes.

> **Note:** Views that receive and edit text must conform to the `NSTextInput` protocol. Adopting this protocol allows a custom view to interact properly with the text input management system. The Application Kit classes `NSText` and `NSTextView` implement `NSTextInput`, so if you subclass these classes you get the protocol conformance "for free."

In handling a key event in `keyDown:`, a view object that expects to insert text first determines whether the character or characters of the `NSEvent` object represent a keyboard action. If they do, it sends the associated action message; if they don't, it inserts the characters as text. Specifically, the view can do one of two things in its implementation:

- It can extract the event object's characters using the `characters` method of `NSEvent` and interpret these to see if they are associated with a known keyboard action. If they are, it invokes the appropriate action method in itself or a superview. This approach is discouraged.
- It can pass the event to Cocoa's text input management system by invoking the `NSResponder` method `interpretKeyEvents:`. The input management system checks the pressed key against entries in all relevant key-binding dictionaries and, if there is a match, sends a `doCommandBySelector:` message back to the view. Otherwise, it sends an `insertText:` message back to the view, and the view implements this method to extract and display the text.

Listing 5-1 shows how the second approach might look in code.

Listing 5-1 Using the input management system to interpret a key event

```
- (void)keyDown:(NSEvent *)theEvent {
    [self interpretKeyEvents:[NSArray arrayWithObject:theEvent]];
}

// The following action methods are declared in NSResponder.h
- (void)insertTab:(id)sender {
```

```
    if ([[self window] firstResponder] == self) {
        [[self window] selectNextKeyView:self];
    }
}

- (void)insertBacktab:(id)sender {
    if ([[self window] firstResponder] == self) {
        [[self window] selectPreviousKeyView:self];
    }
}

- (void)insertText:(id)string {
    [super insertText:string];  // have superclass insert it
}
```

Note that this example includes an override of `insertText:` that simply invokes the superclass implementation. This is done to clarify the role of the input manager but is not really necessary. The default (`NSResponder`) implementation of `doCommandBySelector:` determines if the view responds to the keyboard-action selector and, if the view does respond, it invokes the action method; if the view doesn't respond, `doCommandBySelector:` is sent to the next responder (and so on up the responder chain). Therefore, a view should only implement the action methods corresponding to the actions that it wants to handle. Another implication of input-manager behavior is that if the key-bindings dictionary matches a physical keystroke with a keyboard action, the responder object simply needs to override the associated action method to handle that keystroke. For example, to handle the default binding of the Escape key, the responder would override the `cancelOperation:` method of `NSResponder`.

A case in point of a superview handling a keyboard-action message initiated by a subview is the way the `NSScrollView` object handles page-down commands. This scroll-view object is a compound object consisting of a document view, a clip view (an `NSClipView` object) and a scroller (an `NSScroller` object). Because it is the containing or coordinating object, the `NSScrollView` object is the superview of all other objects in this grouping. Now say your custom view is the document view of the scroll view. If you implement `keyDown:` to send `interpretKeyEvents:` to the input manager but do not implement the `scrollPageDown:` action method, the document view will still be scrolled within the scroll view when the user presses the Page Down key (or whatever key binding is in effect for that function). This happens because each next responder in the responder chain is queried to see if it responds to `scrollPageDown:`. The `NSScrollView` class provides a default implementation, so this implementation is invoked.

Applications other that those that deal with text can use the input management system to their benefit. For example, a custom view in a drawing application might use arrow keys to "nudge" graphical objects precise distances. In the standard key bindings dictionary the arrow keys are bound to the `moveUp:`, `moveDown:`, `moveLeft:`, and `moveRight:` methods of `NSResponder`. So code similar to that shown in Listing 5-2 would work to nudge the graphical objects around.

Listing 5-2 Handling arrow-key characters using the input management system

```
- (void)keyDown:(NSEvent *)theEvent {
    // Arrow keys are associated with the numeric keypad
    if ([theEvent modifierFlags] & NSNumericPadKeyMask) {
        [self interpretKeyEvents:[NSArray arrayWithObject:theEvent]];
    } else {
        [super keyDown:theEvent];
    }
}

-(IBAction)moveUp:(id)sender
{
    [self offsetLocationByX:0 andY: 10.0];
    [[self window] invalidateCursorRectsForView:self];
}

-(IBAction)moveDown:(id)sender
{
    [self offsetLocationByX:0 andY:-10.0];
    [[self window] invalidateCursorRectsForView:self];
}

-(IBAction)moveLeft:(id)sender
{
    [self offsetLocationByX:-10.0 andY:0.0];
    [[self window] invalidateCursorRectsForView:self];
}

-(IBAction)moveRight:(id)sender
```

```
{
    [self offsetLocationByX:10.0 andY:0.0];
    [[self window] invalidateCursorRectsForView:self];
}
```

In most instances, the `interpretKeyEvents:` approach is preferable to the interpret-yourself approach. This recommendation is particularly true for those custom views in applications such as word processors and graphical editors that do the lion's share of the work. The major factor favoring the use of the text input management system is that, with it, you don't need to hard-wire function to physical key. What if the user is using a portable computer that is lacking a function key hard-wired by the application? The better, more flexible approach is to specify alternative key bindings in a dictionary. Another advantage of the text input management system is that it allows key events to be interpreted as text not directly available on the keyboard, such as Kanji and some accented characters.

> **Note:** For more information on key bindings and the text input system, see "Text System Defaults and Key Bindings" (page 105).

Specially Interpreting Keystrokes

Although using the text input management system is advantageous, you can interpret physical keys yourself in `keyDown:` and handle them in an application-specific way. The `NSEvent` class declares dozens of constants that identify particular keys either by key symbol or key function; "Constants" in *NSEvent Class Reference* in the reference documentation for that class describes these constants. Listing 5-3 shows a sampling.

Listing 5-3 Some key constants defined by NSResponder

```
enum {
    NSUpArrowFunctionKey           = 0xF700,
    NSDownArrowFunctionKey         = 0xF701,
    NSLeftArrowFunctionKey         = 0xF702,
    NSRightArrowFunctionKey        = 0xF703,
    NSF1FunctionKey                = 0xF704,
    NSF2FunctionKey                = 0xF705,
    NSF3FunctionKey                = 0xF706,
    // other constants here
    NSUndoFunctionKey              = 0xF743,
    NSRedoFunctionKey              = 0xF744,
```

```
    NSFindFunctionKey               = 0xF745,
    NSHelpFunctionKey               = 0xF746,
    NSModeSwitchFunctionKey         = 0xF747
};
```

Also, the text system defines constants representing commonly-used Unicode characters, such as tab, delete, and carriage return. For a list of these constants, see "Constants" in *NSText Class Reference*.

In your implementation of `keyDown:` you can compare one of these constants to the character data of the key-event object to determine if a certain key was pressed and then act accordingly. As you may recall, the `characters` or `charactersIgnoringModifiers` methods return an `NSString` object for a key value instead of a character because a keystroke might generate multiple characters. (In fact, these methods could even return an empty string if a dead key—a key with no character mapped to it—is pressed.) If your implementation of `keyDown:` is handling a single-character key value, such as an arrow key, you can examine the length of the returned string and, if it is a single character, access that character using the `NSString` method `characterAtIndex:` with an index of 0. Then test that character against one of the `NSResponder` constants.

Listing 5-4 shows how you might perform the same graphical-object "nudging" as done in Listing 5-2 (page 70), but this time the responder object itself determines whether an arrow key was pressed.

Listing 5-4 Handling arrow-key characters by interpreting the physical key

```
- (void)keyDown:(NSEvent *)theEvent {


    if ([theEvent modifierFlags] & NSNumericPadKeyMask) { // arrow keys have this
 mask
        NSString *theArrow = [theEvent charactersIgnoringModifiers];
        unichar keyChar = 0;
        if ( [theArrow length] == 0 )
            return;            // reject dead keys
        if ( [theArrow length] == 1 ) {
            keyChar = [theArrow characterAtIndex:0];
            if ( keyChar == NSLeftArrowFunctionKey ) {
                [self offsetLocationByX:-10.0 andY:0.0];
                [[self window] invalidateCursorRectsForView:self];
                return;
            }
```

```
            if ( keyChar == NSRightArrowFunctionKey ) {
                [self offsetLocationByX:10.0 andY:0.0];
                [[self window] invalidateCursorRectsForView:self];
                return;
            }
            if ( keyChar == NSUpArrowFunctionKey ) {
                [self offsetLocationByX:0 andY: 10.0];
                [[self window] invalidateCursorRectsForView:self];
                return;
            }
            if ( keyChar == NSDownArrowFunctionKey ) {
                [self offsetLocationByX:0 andY:-10.0];
                [[self window] invalidateCursorRectsForView:self];
                return;
            }
            [super keyDown:theEvent];
        }
    }
    [super keyDown:theEvent];
}
```

You can also convert an `NSResponder` constant to a string object and then compare that object to the value returned by `characters` or `charactersIgnoringModifiers`, as in this example:

```
        unichar la = NSLeftArrowFunctionKey;
        NSString *laStr = [[[NSString alloc] initWithCharacters:&la length:1]
autorelease];
        if ([theArrow isEqual:laStr]) {
            [self offsetLocationByX:-10.0 andY:0.0];
            [[self window] invalidateCursorRectsForView:self];
            return;
        }
```

However, this approach is more memory-intensive.

Handling Key Equivalents

A key equivalent is a character bound to some view in a window. This binding causes that view to perform a specified action when the user types that character, typically while pressing a modifier key (in most cases the Command key). A key equivalent must be a character that can be typed with no modifier keys, or with Shift only.

An application routes a key-equivalent event by sending it first down the view hierarchy of a window. The global `NSApplication` object dispatches events it recognizes as potential key equivalents (based on the presence of modifier flags) in its `sendEvent:` method. It sends a `performKeyEquivalent:` message to the key `NSWindow` object. This object passes key equivalents down its view hierarchy by invoking the `NSView` default implementation of `performKeyEquivalent:`, which forwards the message to each of its subviews (including contextual and pop-up menus) until one responds `YES`; if none does, it returns `NO`. If no object in the view hierarchy handles the key equivalent, `NSApp` then sends `performKeyEquivalent:` to the menus in the menu bar. `NSWindow` subclasses are discouraged from overriding `performKeyEquivalent:`.

> **Note:** Beginning with OS X v10.5, if a key equivalent is not recognized, `NSWindow` sends it as an `NSKeyDown` event to the first responder. This behavior enables custom key-binding entries with Command-key modifiers. In addition, `NSApplication` sends a Control-key event to the key window via `performKeyEquivalent:` before sending it as an `NSKeyDown` event through the responder chain. This behavior allows more reliable use of Control-key events as menu key equivalents.

Some Cocoa classes, such as `NSButton`, `NSMenu`, `NSMatrix`, and `NSSavePanel`, provide default implementations of `performKeyEquivalent:`. For example, you can set the Return key as the key equivalent of an `NSButton` object and, when this key is pressed, the button acts as if it has been clicked. However, subclasses of other Application Kit classes (including custom views) need to provide their own implementations of `performKeyEquivalent:`. The implementation should extract the characters for a key equivalent from the passed-in `NSEvent` object using the `charactersIgnoringModifiers` method and then examine them to determine if they are a key equivalent it recognizes. It handles the key equivalent much as it would handle a key event in `keyDown:` (see "Overriding the keyDown: Method" (page 67)). After handling the key equivalent, the implementation should return `YES`. If it doesn't handle the key equivalent, it should either invoke the superclass implementation of `performKeyEquivalent:` or (if you know the superclass doesn't handle the key equivalent) return `NO` to indicate that the key equivalent should be passed further down the view hierarchy or to the menus in the menu bar.

Keyboard Interface Control

The Cocoa event-dispatch architecture treats certain key events as commands to move control focus to a different user-interface object in a window, to simulate a mouse click on an object, to dismiss modal windows, and to make selections in objects that allow selections. This capability is called keyboard interface control. Most of the user-interface objects involved in keyboard interface control are `NSControl` objects, but objects that aren't controls can participate as well. When an object has control focus, the Application Kit draws a light-blue key-focus ring around the object's border. If full keyboard access is enabled, the keys listed in Table 5-1 have the stated effect.

Table 5-1 Keys used in keyboard interface control

Key	Effect
Tab	Move to next key view.
Shift-Tab	Move to previous key view.
Space	Select, as with mouse click in a check box (for example), or toggle state. In selection lists, selects or deselects highlighted item.
Arrow keys	Move within compound view, such as `NSForm` objects.
Control-Tab (Control-Shift-Tab)	Go to next (previous) key view from views where tab characters have other significance (for example, `NSTextView` objects).
Option or Shift	Extend the selection, not affecting other selected items.

Some objects found on Interface Builder palettes do not participate in keyboard interface control, such as `NSImageView`, `WebView`, and `PDFView` objects.

In addition to the key view loop, a window can have a default button cell, which uses the Return (or Enter) key as its key equivalent. Programmatically, you can send `setDefaultButtonCell:` to an `NSWindow` object to set this button cell; you can also set it in Interface Builder by setting a button cell's key equivalent to '`\r`' in the Attributes pane of the Get Info window. The default button cell draws itself as a focal element for keyboard interface control unless another button cell is focused on. In this case, it temporarily draws itself as normal and disables its key equivalent. The Escape key is another default key for a keyboard interface control in a window; it immediately aborts a modal loop.

The user-interface objects that are connected together in a window make up the window's key view loop. A key view loop is a sequence of `NSView` objects connected to each other through their `nextKeyView` property (the `previousKeyView` property when going in reverse direction). The last view in this sequence "loops" back

to the first view. By default, NSWindow assigns an initial first responder and constructs a key view loop with the objects it finds. If you want greater control over the key view loop you should set it up using Interface Builder. See the Help pages for Interface Builder for details of the procedure.

For its instances to participate in key-view loops, a custom view must return YES from acceptsFirstResponder. By doing so, it affects the value returned by the canBecomeKeyView method. The acceptsFirstResponder method controls whether a responder accepts first responder status when its window asks it to (that is, when makeFirstResponder: is called with the responder as the parameter). The canBecomeKeyView method controls whether the Application Kit allows tabbing to a view. It calls acceptsFirstResponder, but it also checks for other information before determining the value to return, such as whether the view is hidden and whether full keyboard access is on. The canBecomeKeyView method is rarely overridden while acceptsFirstResponder is frequently overridden.

The NSView and NSWindow classes define a number of methods for setting up and traversing the key view loop programmatically. Table 5-2 lists some of the more useful ones.

Table 5-2 Some key-view loop methods

nextKeyView (NSView) previousKeyView (NSView)	Returns the next and previous view objects in the key view loop.
setNextKeyView: (NSView)	Sets the next key view in the loop.
selectNextKeyView: (NSWindow) selectPreviousKeyView: (NSWindow)	Searches the view hierarchy for a candidate next (previous) key view and, if it finds one, makes it the first responder.
canBecomeKeyView (NSView)	Returns whether the receiver can become a key view.
nextValidKeyView (NSView) previousValidKeyView (NSView)	Returns the closest view object in the key view loop that follows the receiver and accepts first responder status.

The code in Listing 5-5 illustrates how one might use some of these methods to manipulate the key-view loop.

Listing 5-5 Manipulating the key-view loop

```
- (void)textDidEndEditing:(NSNotification *)notification {
    NSTextView *text = [notification object];
    unsigned whyEnd = [[[notification userInfo] objectForKey:@"NSTextMovement"]
unsignedIntValue];
    NSTextView *newKeyView = text;
```

```
    // Unscroll the previous text.
    [text scrollRangeToVisible:NSMakeRange(0, 0)];

    if (whyEnd == NSTabTextMovement) {
        newKeyView = (NSTextView *)[text nextKeyView];
    } else if (whyEnd == NSBacktabTextMovement) {
        newKeyView = (NSTextView *)[text previousKeyView];
    }

    // Set the new key view and select its whole contents.
    [[text window] makeFirstResponder:newKeyView];
    [newKeyView setSelectedRange:NSMakeRange(0, [[newKeyView textStorage] length])];
}
```

Using Tracking-Area Objects

An instance of the `NSTrackingArea` class defines an area of a view that is responsive to the movements of the mouse. When the mouse cursor enters this area, moves around in it, and leaves it, the Application Kit sends (depending on the options specified) mouse-tracking, mouse-moved, and cursor-update messages to a designated object.

> **Important:** The `NSTrackingArea` class was introduced in OS X v10.5. For a discussion of the API used for mouse tracking and cursor updating in earlier releases, see the appendix "Using Tracking-Area Objects" (page 78). Also see "Compatibility Issues" (page 83), below, for a discussion of the current behavior of legacy methods.

The Application Kit sends the mouse-tracking messages `mouseEntered:` and `mouseExited:` to an object when the mouse cursor enters and exits a region of a window. Mouse tracking enables the view owning the region to respond, for example, by drawing a highlight color or displaying a tool tip. The Application Kit also sends `mouseMoved:` messages to the object if `NSMouseMoved` types of events are requested. Cursor-update events are a special kind of mouse-tracking event that the Application Kit handles automatically. When the mouse pointer enters a cursor rectangle, the Application Kit displays a cursor image appropriate to the type of view under the rectangle; for example, when a mouse pointer enters a text view, an I-beam cursor is displayed.

The sections in this chapter describe how to create `NSTrackingArea` objects, attach them to views, respond to related events, and manage the objects when changes in view geometry occur.

Creating an NSTrackingArea Object

An `NSTrackingArea` object defines a region of a view that is sensitive to the movements of the mouse. When the mouse enters, moves about, and exits that region, the Application Kit sends mouse-tracking, mouse-moved, and cursor-update messages. The region is a rectangle specified in the local coordinate system of its associated view. The recipient of the messages (the `owner`) is specified when the tracking-area object is created; although the owner can be any object, it is often the view associated with the tracking-area object.

When you create an `NSTrackingArea` object you must specify one or more options. These options, which are enumerated constants declared by the class, configure various aspects of tracking-area behavior. They fall into three general categories:

- **The type of event message sent**

You can request `mouseEntered:` and `mouseExited:` messages (`NSTrackingMouseEnteredAndExited`); you can request `mouseMoved:` messages (`NSTrackingMouseMoved`); and you can request `cursorUpdate:` messages (`NSTrackingCursorUpdate`).

You are not limited to a single option from this set; you can perform a bitwise-OR operation to request multiple types of messages.

- **The active scope of tracking-area messages**

  You must specify *one* of the following options to request when the tracking area should actively generate events:

  - When the associated view is first responder (`NSTrackingActiveWhenFirstResponder`)
  - When the associated view is in the key window (`NSTrackingActiveInKeyWindow`)
  - When the associated view is in the active application (`NSTrackingActiveInActiveApp`)
  - At all times regardless of application activation (`NSTrackingActiveAlways`)

- **Refinements of tracking-area behavior**

  You can request that the first message be sent when the mouse cursor first leaves the tracking area (`NSTrackingAssumeInside`); you can request the tracking area to be the same as the visible rectangle of the associated view (`NSTrackingInVisibleRect`); and you can request that mouse drags into and out of the tracking area generate `mouseEntered:` and `mouseExited:` events (`NSTrackingEnabledDuringMouseDrag`).

  You are not limited to a single option from this set; you can perform a bitwise-OR operation to request multiple refinements on behavior.

You initialize an allocated instance of `NSTrackingArea` with the `initWithRect:options:owner:userInfo:` method. After creating the object, you must associate it with a view by invoking the `NSView` method `addTrackingArea:`. You can create an `NSTrackingArea` instance and add it to a view at any point because (unlike the mouse-tracking API in earlier releases), successful creation does not depend on the view being added to a window. Listing 6-1 shows the creation and addition of an `NSTrackingArea` instance in a custom view's `initWithFrame:` method; in this case, the owning view is requesting that the Application Kit send `mouseEntered:`, `mouseExited:` and `mouseMoved:` messages whenever its window is the key window.

Listing 6-1 Initializing an `NSTrackingArea` instance

```
- (id)initWithFrame:(NSRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        trackingArea = [[NSTrackingArea alloc] initWithRect:eyeBox
            options: (NSTrackingMouseEnteredAndExited | NSTrackingMouseMoved |
NSTrackingActiveInKeyWindow )
```

```
            owner:self userInfo:nil];
        [self addTrackingArea:trackingArea];
    }
    return self;
}
```

The last parameter of `initWithRect:options:owner:userInfo:` is a dictionary that you can use to pass arbitrary data to the recipient of mouse-tracking and cursor-update messages. The receiving object can access the dictionary by sending `userData` to the `NSEvent` object passed into the `mouseEntered:`, `mouseExited:`, or `cursorUpdate:` method. Sending `userData` messages in `mouseMoved:` methods causes an assertion failure. (In the example in Listing 6-1, the `userInfo` parameter is set to `nil`.)

Tracking rectangle bounds are inclusive for the top and left edges, but not for the bottom and right edges. Thus, if you have a unflipped view with a tracking rectangle covering its bounds, and the view's frame has the geometry `frame.origin = (100, 100), frame.size = (200, 200)`, then the area for which the tracking rectangle is active is `frame.origin = (100, 101), frame.size = (199, 199)`, in frame coordinates.

Managing a Tracking-Area Object

Because `NSTrackingArea` objects are owned by their views, the Application Kit can automatically recompute the tracking-area regions when the view is added or removed from a window or when the view changes position within its window. But in situations where the Application Kit cannot recompute an affected tracking area (or areas), it sends `updateTrackingAreas` to the associated view, asking it to recompute and reset the areas. One such situation is when a change in view location affects the view's visible rectangle (`visibleRect`)—unless the `NSTrackingArea` object for that view was created with the `NSTrackingInVisibleRect` option, in which case the Application Kit handles the re-computation. Note that the Application Kit sends `updateTrackingAreas` to every view whether it has a tracking area or not.

You can override the `updateTrackingAreas` method as shown in Listing 6-2 to remove the current tracking areas from their views, release them, and then add new `NSTrackingArea` objects to the same views with recomputed regions.

Listing 6-2 Resetting a tracking-area object

```
- (void)updateTrackingAreas {
    NSRect eyeBox;
```

```
    [self removeTrackingArea:trackingArea];
    [trackingArea release];
    eyeBox = [self resetEye];
    trackingArea = [[NSTrackingArea alloc] initWithRect:eyeBox
        options: (NSTrackingMouseEnteredAndExited | NSTrackingMouseMoved |
NSTrackingActiveInKeyWindow)
        owner:self userInfo:nil];
    [self addTrackingArea:trackingArea];
}
```

If your class is not a custom view class, you can register your class instance as an observer for the notification `NSViewFrameDidChangeNotification` and have it reestablish the tracking rectangles on receiving the notification.

Responding to Mouse-Tracking Events

The owner of an `NSTrackingArea` object created with the `NSTrackingMouseEnteredAndExited` option receives a `mouseEntered:` whenever the mouse cursor enters the region of a view defined by the tracking-area object; subsequently, it receives a `mouseExited:` messages when the mouse leaves that region. If the `NSTrackingMouseMoved` option is also specified for the tracking-area object, the owner also receives one or more `mouseMoved:` messages between each `mouseEntered:` and `mouseExited:` message. You override the corresponding `NSResponder` methods to handle these messages to perform such tasks as highlighting the view, displaying a custom tool tip, or displaying related information in another view.

Important: The proper order of mouse-entered and mouse-exited events received by tracking-area objects in an application cannot be guaranteed. For example, if you move the mouse cursor from one tracking area to another tracking area and back, the order of events (as messages) could be: `mouseEntered:`, `mouseEntered:`, `mouseExited:`, `mouseExited:`.

The tracking code in Listing 6-2 is used in making an "eyeball" follow the movement of the mouse pointer when it enters a tracking rectangle.

Listing 6-3 Handling mouse-entered, mouse-moved, and mouse-exited events

```
- (void)mouseEntered:(NSEvent *)theEvent {
    NSPoint eyeCenter = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    eyeBox = NSMakeRect((eyeCenter.x-10.0), (eyeCenter.y-10.0), 20.0, 20.0);
```

```
    [self setNeedsDisplayInRect:eyeBox];
    [self displayIfNeeded];
}

- (void)mouseMoved:(NSEvent *)theEvent {
   NSPoint eyeCenter = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    eyeBox = NSMakeRect((eyeCenter.x-10.0), (eyeCenter.y-10.0), 20.0, 20.0);
    [self setNeedsDisplayInRect:eyeBox];
    [self displayIfNeeded];
}

- (void)mouseExited:(NSEvent *)theEvent {
    [self resetEye];
    [self setNeedsDisplayInRect:eyeBox];
    [self displayIfNeeded];
}
```

Just as you can with mouse-down and mouse-up messages, you can query the passed-in `NSEvent` objects to get information related to the event.

Managing Cursor-Update Events

One common use of `NSTrackingArea` objects is to change the cursor image over different types of views. Text, for example, typically requires an I-beam cursor. Many Application Kit classes provide cursor images appropriate to their view instances; you get this behavior "for free." However, you may want to specify a specific (or different) cursor image for instances of your custom view subclasses.

You change the cursor image for your view in an override of the `NSResponder` method `cursorUpdate:`. To receive this message, you must create an `NSTrackingArea` object by invoking the `initWithRect:options:owner:userInfo:` initializer with an option of `NSTrackingCursorUpdate` (along with any other desired options). Then add the created object to a view with `addTrackingArea:`. Thereafter, the entry of the mouse into the tracking area generates an `NSCursorUpdate` event; the `NSWindow` object handles this event by sending a `cursorUpdate:` message to the owner of the tracking area (which is typically the view itself). The implementation of `cursorUpdate:` should use the appropriate `NSCursor` methods to set a standard or custom cursor image.

Cursor rectangles may overlap or be completely nested, one within the other. Arbitration of cursor updates follows the normal responder chain mechanism. The cursor rectangle of the view under the mouse cursor first receives the `cursorUpdate:` message. It may either display a cursor or pass the message up the responder chain, where a view with an overlapped cursor rectangle can then respond to the message.

Listing 6-4 shows an implementation of `cursorUpdate:` that sets the cursor of the view to a cross-hair image. Note that it is not necessary to reset the cursor image back to what it was when the mouse exits the tracking area. The Application Kit handles this automatically for you by sending a `cursorUpdate:` message to the view over which the mouse cursor moves as it exits the cursor rectangle.

Listing 6-4 Handling a cursor-update event

```
-(void)cursorUpdate:(NSEvent *)theEvent
{
    [[NSCursor crosshairCursor] set];
}
```

If the responder owning the tracking-area object does not implement the `cursorUpdate:` method, the default implementation forwards the message up the responder chain. If the responder implements `cursorUpdate:` but decides not to handle the current event, it should invoke the superclass implementation of `cursorUpdate:`.

As with any other kind of `NSTrackingArea` object, you might occasionally need to recompute and recreate a tracking-area object used for cursor updates when the associated view has changes in its location or size. See "Managing a Tracking-Area Object" (page 80) for more information.

Compatibility Issues

Beginning with OS X v10.5, the `NSTrackingArea` class and the related `NSView` methods `addTrackingArea:`, `removeTrackingArea:`, `updateTrackingAreas`, and `trackingAreas` replace the following methods of `NSView`, which are considered legacy API but remain supported for compatibility:

`addTrackingRect: owner:userData: assumeInside:`
`removeTrackingRect:`
`addCursorRect: cursor:`
`removeCursorRect: cursor:`
`discardCursorRects`

`resetCursorRects`

The `NSTrackingRectTag` type, returned by the `addTrackingRect:owner:userData:assumeInside:` and passed into the `removeTrackingRect:` methods, is also a legacy type. Internally this type is treated the same as `NSTrackingArea *`.

The underlying implementation for the legacy methods is based on `NSTrackingArea`, resulting in the following implications:

- An invocation of the `addTrackingRect:owner:userData:assumeInside:` method creates an `NSTrackingArea` object with the options `NSTrackingMouseEnteredAndExited` and `NSTrackingActiveAlways` set. It also includes the `NSTrackingAssumeInside` option if the last parameter of the function is `YES`. After creating the object, it adds it to the receiver (`addTrackingArea:`) and returns it as a tag.
- The `resetCursorRects` method is invoked after `updateTrackingAreas`.

In addition, the following `NSWindow` cursor-related methods are legacy API, but maintained for compatibility:

`discardCursorRects`
`invalidateCursorRectsForView:`
`resetCursorRects`

Handling Tablet Events

The following sections discuss issues related to the handling of tablet events.

Packaging of Tablet Events

Tablet device drivers package low-level events as either native tablet events or as mouse events, usually depending on whether they are proximity events or pointer events. Proximity events are always native tablet events. The Application Kit declares (in `NSEvent.h`) the following event-type constants for native tablet events:

```
typedef enum _NSEventType {
    // ...
    NSTabletPoint     = 23,
    NSTabletProximity = 24,
    // ...
} NSEventType;
```

Drivers almost always package tablet-pointer events as subtypes of mouse events. The Application Kit declares the following constants for tablet subtypes of all event types related to mouse event (`NSLeftMouseDown`, `NSRightMouseUp`, `NSMouseMoved`, and so on):

```
enum {
    NSMouseEventSubtype           = NX_SUBTYPE_DEFAULT,
    NSTabletPointEventSubtype      = NX_SUBTYPE_TABLET_POINT,
    NSTabletProximityEventSubtype = NX_SUBTYPE_TABLET_PROXIMITY
};
```

Under a few exceptional conditions drivers may package a low-level tablet-pointer event as a `NSTabletPoint` event type instead of a mouse-event subtype. These include the following:

- During the interval between a mouse-down (that is, stylus-down) and subsequent dragging events when, for example, only pressure is changing.

- When there are two concurrently active pointing devices, the one not moving the cursor generates `NSTabletPoint` events.
- If, for some reason, the tablet driver is told not to move the cursor, the driver packages tablet events in native form.

For this reason, it is recommended that your code check for tablet-pointer events delivered both as native event types and as mouse subtypes.

Tablet Events and the Responder Chain

Like any NSEvent object, tablet events are routed up the responder chain until they are handled. Responder objects in an application (that is, objects that inherit from NSResponder) can override the appropriate NSResponder methods and handle the NSEvent object that is passed to them in that method. Or they can pass the event on to the next responder in the chain. (If you don't override one of these method, the event automatically goes to the next responder.

An application that intends to handle tablet events should override at least five NSResponder methods:

- `tabletProximity:` and `tabletPoint:`

  Implement these methods to handle native proximity and pointer tablet events.

- `mouseDown:`, `mouseDragged:`, and `mouseUp:`

  implement these methods to handle mouse events with a subtype of `NSTabletPointEventSubtype` or `NSTabletProximityEventSubtype`.

A recommended approach is to funnel tablet events in these methods to two common handlers, one for proximity events and the other for pointer events. If you have objects in your application that are not in the responder chain and you want these objects to know about tablet events as they arrive, you could implement your event-handler routine so that it posts a notification to all interested observers.

Handling Trackpad Events

When users touch and move their fingers on the trackpads of the MacBook Air and more recent models of the MacBook Pro, OS X generates multi-touch events, gesture events, and mouse events. The trackpad hardware includes built-in support for interpreting common gestures and for mapping movements of a finger to mouse events. In addition, the operating system provides default handling of gestures; you can observe how OS X handles these gestures in the Trackpad system preference. You can also create applications that receive and respond to gestures and multi-touch events in distinctive ways.

An application should not rely on gesture-event or touch-event handling as the sole mechanism for interpreting user actions for any critical feature. Users could be using a desktop computer or, if they are using a laptop, might not be using the trackpad at all—for example, they could have a USB mouse plugged into the laptop. Touch-event features should supplement the conventional way of conveying user commands.

> **Note:** Support for gestures was introduced in OS X v10.5.2. The `NSTouch` class and related programmatic interfaces for touch-event handling were introduced in OS X v10.6 (Snow Leopard).

Gesture events are a species of multi-touch events because they're based on an interpretation of a sequence of touches. In other words, gestures are a series of multi-touch events recognized by the trackpad as constituting a gesture. Gestures and touch events require you to adopt a different approach for handling them. To understand these approaches it's useful to know how and when the events are generated, how they are delivered to your application, and what kind of information they contain about the actual touches on the trackpad.

Gestures are Touch Movements Interpreted By the Trackpad

Gestures are particular movements of fingers on a touch-sensitive surface, such as a trackpad's, that have a conventional significance. The trackpad driver interprets three of these movements as specific gestures:

- Pinching movements (in or out) are gestures meaning zoom out or zoom in (also called magnification).
- Two fingers moving in opposite semicircles is a gesture meaning rotate.
- Three fingers brushing across the trackpad surface in a common direction is a swipe gesture.
- Two fingers moving vertically or horizontally is a scroll gesture.

The system delivers low-level events representing specific gestures to the active application, where they're packaged as `NSEvent` objects and placed in the application's event queue. They go along the same path as mouse events for delivery to the view under the mouse pointer. An `NSWindow` object delivers a gesture event to the view by calling the appropriate `NSResponder` method for the gesture: `magnifyWithEvent:`, `rotateWithEvent:`, or `swipeWithEvent:`. If the view does not handle a gesture, it travels up the responder chain until another object handles it or until it is discarded.

Types of Gesture Events and Gesture Sequences

The AppKit framework, in `NSEvent.h`, declares the `NSEventType` constants for gestures shown in Listing 8-2.

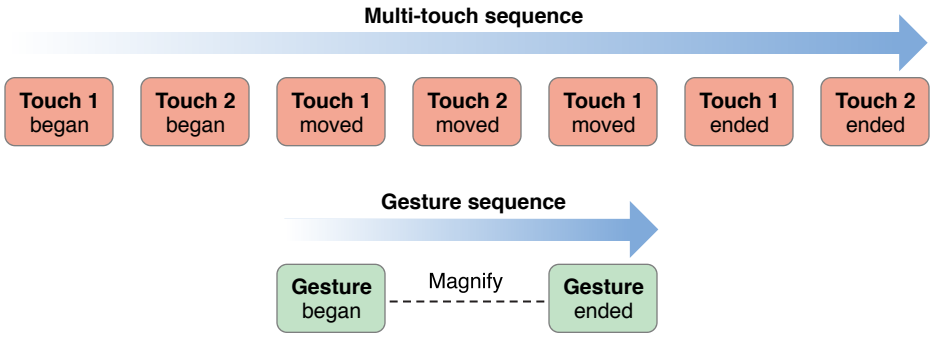
Listing 8-1 Constants for gesture events in the AppKit framework

```
NSEventTypeGesture          = 29,
NSEventTypeMagnify          = 30,
NSEventTypeSwipe            = 31,
NSEventTypeRotate           = 18,
NSEventTypeBeginGesture     = 19,
NSEventTypeEndGesture       = 20
```

As users touch and move their fingers across the trackpad, the trackpad driver generates a **gesture sequence**, which is roughly concurrent with the multi-touch sequence described in "Touch Events Represent Fingers on the Trackpad" (page 93) but contained within it. A gesture sequence begins when the driver first detects a gesture and ends when the driver determines the gesture has ended. Events of type `NSEventTypeBeginGesture` and `NSEventTypeEndGesture` mark the start and end points of a gesture. In most cases, you do not need to be aware of the gesture sequence or these event types; you only need to implement the `NSResponder` method for handling the specific gesture. However, if you want to commit a

change related to a specific gesture, such as register an undo operation or perform detailed drawing, you may implement the `NSResponder` methods `beginGestureWithEvent:` and `endGestureWithEvent:` in addition to the methods for specific gestures.

Figure 8-1 A gesture sequence within a multi-touch sequence



If you do this, however, you should be aware of some of the behavioral peculiarities of gesture detection. First, multiple gesture sequences can occur within a single multi-touch sequence. For example, the user might first pinch a view and then swipe it without removing all of her fingers from the trackpad. Moreover, between the begin-gesture and end-gesture events of a gesture sequence, the trackpad driver might first interpret a movement as one gesture and then switch its interpretation to another gesture. Currently, however, this happens only for magnify and rotate gestures. Scroll and swipe gestures, once begun, are locked to that gesture until the gesture ends.

As it generates gestures, the trackpad hardware *might* also emit the touch events that make up the gesture; the system then routes the touch events to the application along with the gesture events. Some trackpads, however, do not support this feature.

The Trackpad preference pane includes an option for scroll gestures: two fingers moving the content view of a scroll view around. Technically, scroll gestures are not specific gestures but mouse events. The trackpad driver wraps a sequence of mouse events affecting the scroll view between a gesture-begin and gesture-end event pair. Other than detecting these events, you have no way to determine if a scroll event was generated by a touch.

> **Note:** The `NSEventTypeGesture` constant shown in Listing 8-2 represents a "generic" gesture, which is generated by the trackpad driver when movements on the trackpad do not result in a specific gesture. You may safely ignore this event type.

Handling Gesture Events

To handle an event for a specific gesture—a rotation, pinching, or swipe movement—implement in your custom view the appropriate `NSResponder` method—in this case `rotateWithEvent:`, `magnifyWithEvent:`, or `swipeWithEvent:`. Unlike the procedure for handling multi-touch events, your custom view does not have to "opt-in". When the mouse point is over your view and the user makes a gesture, the corresponding `NSResponder` method is called to handle the event. It is also called if the views earlier in the responder chain do not handle the event.

Each of the gesture-handling methods has an `NSEvent` parameter. You may query the event object for information pertinent to the gesture. For the three recognized gestures, the following event-object attributes have special importance:

- Zoom in or out (`NSEventTypeMagnify`)—The `magnification` accessor method returns a floating-point (`CGFloat`) value representing a factor of magnification.
- Rotation (`NSEventTypeRotate`)—The `rotation` accessor method returns a floating-point value representing the degrees of rotation, counterclockwise.
- Swipe (`NSEventTypeSwipe`)—The `deltaX` and `deltaY` accessor methods return the direction of the swipe as a floating-point (`CGFloat`) value. A non-zero `deltaX` value represents a horizontal swipe; -1 indicates a swipe-right and 1 indicates a swipe-left. A non-0 deltaY represent a vertical swipe; -1 indicates a swipe-down and 1 indicates a swipe-up.

The rotate and magnify gestures are relative events. That is, each `rotateWithEvent:` and `magnifyWithEvent:` message carries with it the change of rotation or magnification since the last gesture event of that type. For zooming in or out, you add the value from the `magnification` accessor to 1.0 to get the scale factor. For rotation, you add the newest degree of rotation to the view's current rotation value. Listing 8-2 illustrates how you might do this.

Listing 8-2 Handling magnification and rotation gestures

```
- (void)magnifyWithEvent:(NSEvent *)event {
    [resultsField setStringValue:
        [NSString stringWithFormat:@"Magnification value is %f", [event
magnification]]];
    NSSize newSize;
```

```
    newSize.height = self.frame.size.height * ([event magnification] + 1.0);
    newSize.width = self.frame.size.width * ([event magnification] + 1.0);
    [self setFrameSize:newSize];
}

- (void)rotateWithEvent:(NSEvent *)event {
    [resultsField setStringValue:
       [NSString stringWithFormat:@"Rotation in degree is %f", [event rotation]]];
    [self setFrameCenterRotation:([self frameCenterRotation] + [event rotation])];
}
```

To handle a swipe gesture simply requires you to determine the direction of swipe by analyzing the `deltaX` and `deltaY` values. The code in Listing 8-3 responds to swipes by setting a fill color for the implementing view.

Listing 8-3 Handling a swipe gesture

```
- (void)swipeWithEvent:(NSEvent *)event {
    CGFloat x = [event deltaX];
    CGFloat y = [event deltaY];
    if (x != 0) {
        swipeColorValue = (x > 0)  ? SwipeLeftGreen : SwipeRightBlue;
    }
    if (y != 0) {
        swipeColorValue = (y > 0)  ? SwipeUpRed : SwipeDownYellow;
    }
    NSString *direction;
    switch (swipeColorValue) {
        case SwipeLeftGreen:
            direction = @"left";
            break;
        case SwipeRightBlue:
            direction = @"right";
            break;
        case SwipeUpRed:
```

```
            direction = @"up";
            break;
        case SwipeDownYellow:
        default:
            direction = @"down";
            break;
    }
    [resultsField setStringValue:[NSString stringWithFormat:@"Swipe %@", direction]];
    [self setNeedsDisplay:YES];
}
```

You can also query the passed-in `NSEvent` object for other information related to the gesture event, including the location of the mouse pointer in window coordinates (`locationInWindow`), the timestamp of the event, and any modifier keys pressed by the user.

Because a gesture event is derived from a multi-touch sequence, it might seem reasonable to query the `NSEvent` object for its touches by calling `touchesMatchingPhase:inView:`. However, the returned `NSTouch` objects might not be an accurate reflection of the touches currently in play. Thus you should not examine touch objects in the gesture-handling methods. The only reliable set of touch objects is returned in touch-event handling methods such as `touchesBeganWithEvent:`. For more on these methods, see "Touch Events Represent Fingers on the Trackpad" (page 93).

As mentioned in "Types of Gesture Events and Gesture Sequences" (page 88), you can also implement the `beginGestureWithEvent:` and `endGestureWithEvent:` methods to perform actions such as coalescing all the gesture changes between start and end points of the gesture so that you can undo the complete sequence instead of just one step of the sequence.

If your custom view handles one of the API-supported gestures (swipe, magnify, and rotate), your view's implementation is called instead of any other implementation if your view is before the other object in the responder chain. However, there are certain system-wide gestures, such as a four-finger swipe. for which the system implementation takes precedence over any gesture handling an application performs.

Note: You should never handle gesture events by examining events by type in a tracking loop—that is a loop controlled by methods such as `nextEventMatchingMask:untilDate:inMode:dequeue:`.

Touch Events Represent Fingers on the Trackpad

Instead of handling a gesture, you could choose to track and handle the "raw" touches that make up the gesture. But why might you make such a choice? One obvious reason is that OS X does not recognize the particular gesture you are interested in—that is, something other than magnify (pinch in and out), rotate, or swipe. Or you want your view to respond to a system-supported gesture, but you want more information about the gesture than the AppKit framework currently provides; for example, you would like to have anchor points for a zooming operation. Unless you have reasons such as these, you should prefer gestures to raw touch events.

The following sections discuss the multi-touch sequence that delimits a touch event in an abstract sense, point out important touch-event attributes, and show you how to handle touch events.

A Multi-Touch Sequence

When a user touches a trackpad with one or more fingers and moves those fingers over the trackpad, the hardware generates low-level events that represent each of those fingers on the trackpad. The stream of events, as with all type of events, is continuous. However, there is a logical unit of touches that together, represent a **multi-touch sequence**. A multi-touch sequence begins when the user puts one or more fingers on the trackpad. The finger can move in various directions over the trackpad, and additional fingers may touch the trackpad. The multi-touch sequence doesn't end until all of those fingers are lifted from the trackpad.

Within a multi-touch sequence, a finger on the trackpad typically goes through distinct phases:

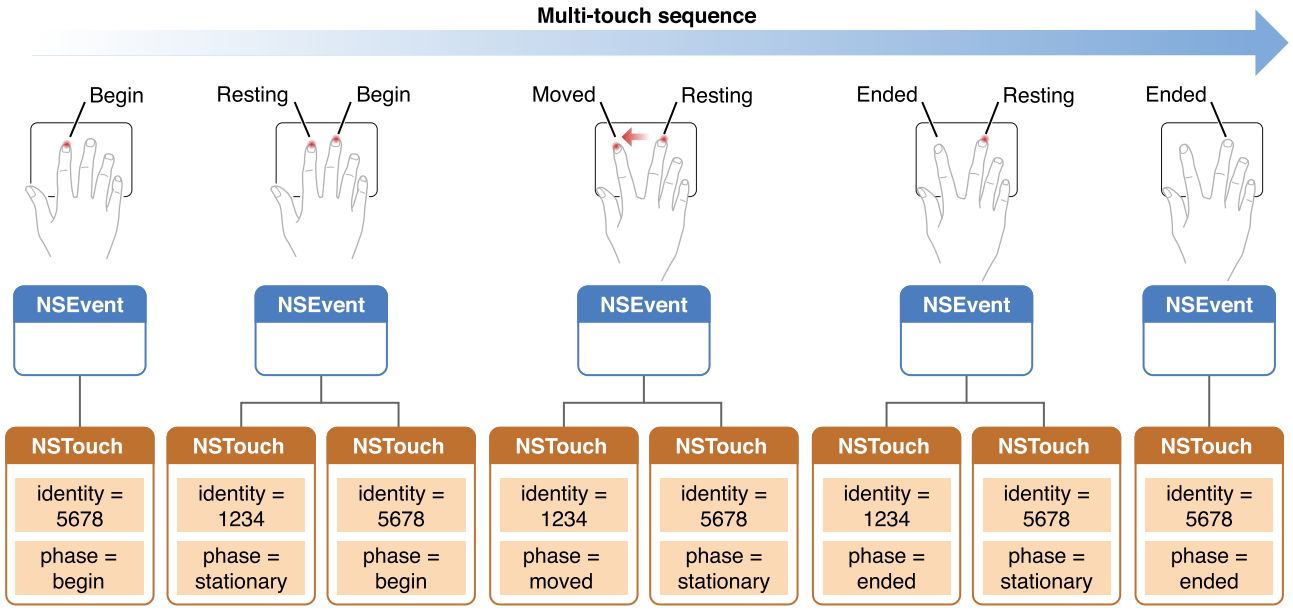
- It touches down on the trackpad.
- It can move in various directions at various speeds.
- It can remain stationary.
- It lifts from the trackpad.

The AppKit framework uses objects of the `NSTouch` class to represent touches through the various phases of a multi-touch sequence. That is, an `NSTouch` object is a snapshot of a particular finger—a touch—on the trackpad in a particular phase. For example, when a touch moves in a certain direction, the AppKit represents it with an `NSTouch` instance; it uses another `NSTouch` object to represent the same finger when it lifts from the trackpad.

> **Note:** This behavior is different from touch objects on iOS, where a `UITouch` object representing a finger on the screen persists through a multi-touch sequence and is mutated throughout that sequence.

Each touch object contains its phase as a property whose value is one of the `NSTouchPhase` constants shown in Listing 8-4 (page 94). It also has an `identity` property that you use to track an `NSTouch` instance through a multi-touch sequence. ("Touch Identity and Other Attributes" (page 95) describes touch identity in more detail.) Figure 8-2 illustrates a multi-touch sequence and the role of these properties in this sequence.

Figure 8-2 Multi-touch sequence on OS X



The AppKit attaches the touches in a multi-touch sequence to the view that is under the mouse pointer when the first finger in the sequence touches down on the trackpad. A touch remains attached to this view until it has ended (that is, the finger lifts) or until the multi-touch sequence is cancelled. Any interpretations of touches in a multi-touch sequence must refer to that view, and that view only. You cannot have touches associated with different views in the same multi-touch sequence. A touch, as represented by a `NSTouch` object, does not have a corresponding screen location or location in its view. But you can determine its changing positions on the trackpad, and from that map a delta value for transforming the view.

Listing 8-4 Constants for touch phases

```
enum {
    NSTouchPhaseBegan             = 1U << 0,
```

```
    NSTouchPhaseMoved           = 1U << 1,
    NSTouchPhaseStationary      = 1U << 2,
    NSTouchPhaseEnded           = 1U << 3,
    NSTouchPhaseCancelled       = 1U << 4,

    NSTouchPhaseTouching        = NSTouchPhaseBegan | NSTouchPhaseMoved |
NSTouchPhaseStationary,
    NSTouchPhaseAny             = NSUIntegerMax
};
typedef NSUInteger NSTouchPhase;
```

Touch Identity and Other Attributes

The `NSTouch` defines a number of properties for its instances. A particularly important property is `identity`. As you may recall from the previous discussion, an application creates an `NSTouch` object to represent the same finger on the trackpad for each phase it goes through in a multi-touch sequence. Although these are different objects, they have the same `identity` object value, enabling you to track changes to a touch throughout a multi-touch sequence. You can determine if two touch objects refer to a particular finger on the trackpad by comparing them with the `isEqual:` method:

```
if ([previousTouchObject.identity isEqual:currentTouchObject.identity]) {
    // object refers to same finger on trackpad...do something appropriate
}
```

Both touch-identity objects and `NSTouch` objects themselves adopt the `NSCopying` protocol. You can therefore copy them. This capability means you can use touch-identity objects as keys in `NSDictionary` collections.

Two other important and related properties of `NSTouch` objects are `normalizedPosition` and `deviceSize`. A touch has no visible location on the screen. (The mouse cursor only identifies the view that first receives touch events if it implements the required methods.) A touch does, however, have a position on the trackpad. The trackpad has a coordinate system defined by an origin of (0,0) in the lower-left corner of the trackpad and a height and width defined by `deviceSize`. Its position in this coordinate system is returned by `normalizedPosition`. Using these two properties you can derive delta values for touch movements and apply these to transformations of the manipulated view. This is illustrated in the code example in Listing 8-7 (page 99).

Resting Touches

The trackpad driver might identify one or more fingers (and thumb) as resting. This “resting” status means that the finger or thumb is physically on the digitizer, but the driver determines that it should not be used for input. For example, with the buttonless trackpads, a user may rest his or her thumb on the bottom section of the trackpad, just as one rests their arms on an arm rest. This digit is ignored as input, and its movement does not move the mouse pointer.

The driver might also transition a touch into or out of a resting status at any time, based on its evaluation. Movement of the resting finger or thumb is not always the determinant. A resting finger that moves might not necessarily transition out of "resting". On the other hand, a resting finger or thumb does not have to physically move at all to transition in and out of "resting".

Even if a touch is flagged as “resting” to denote that its should be ignored, the driver still generates touch data for it and sends an event to the application for each touch phase. By default, these events are not delivered to views and are not included in an event’s set of touches. However, you can turn on this capability by invoking the `NSView` method `setWantsRestingTouches:` with an argument of `YES`. If you are not accepting resting touches, be aware that `NSTouchPhaseBegan` and `NSTouchPhaseEnded` events are generated when a touch transitions from or to a resting state.

Handling Multi-Touch Events

A view by default does not accept touch events. To handle touch events, your custom view must first call the `NSView` method `setAcceptsTouchEvents:` with an argument of `YES`. Then your view must implement the `NSResponder` methods for touch-event handling:

```
- (void)touchesBeganWithEvent:(NSEvent *)event;
- (void)touchesMovedWithEvent:(NSEvent *)event;
- (void)touchesEndedWithEvent:(NSEvent *)event;
- (void)touchesCancelledWithEvent:(NSEvent *)event;
```

For custom subclass of `NSView`, you should implement each of these methods. If you subclass an class that handles touches, you don’t need to implement all these methods methods, but you should call the superclass implementation in the methods you do override.

An application invokes each of these methods on the view when a touch enters a phase—that is, when a finger touches down, when it moves on the trackpad, when it lifts from the trackpad, and when the operating system cancels a multi-touch sequence for some reason. More than one of these following methods could be called for the same event. The application first sends these messages to the view that is under the mouse pointer. If that view does not handle the event, the event is passed up the responder chain.

The single parameter of these responder methods is an `NSEvent` object. You can obtain the set of `NSTouch` objects related to a given phase by sending a `touchesMatchingPhase:inView:` to the event object, passing in the constant for the phase. For example, in the `touchesBeganWithEvent:` method you can get touch objects representing fingers that have just touched down with a call similar to this:

```
NSSet *touches = [event touchesMatchingPhase:NSTouchPhaseBegan inView:self];
```

The operating system calls `touchesCancelledWithEvent:` when an external event—for example, application deactivation—interrupts the current multi-touch sequence. You should implement the `touchesCancelledWithEvent:` method to free resources allocated for touch handling or to reset transient state used in touch handling to initial values.

The remaining code examples are taken from the *LightTable* sample-code project. First, the project declares two static arrays to hold the initial touches and the current touches:

```
NSTouch *_initialTouches[2];
NSTouch *_currentTouches[2];
```

Listing 8-5 illustrates an implementation of the `touchesBeganWithEvent:` method. In this method, the view gets all touches associated with the event and, if there are two touches, stores them in the two static arrays.

Listing 8-5 Handling touches in `touchesBeganWithEvent:`

```
- (void)touchesBeganWithEvent:(NSEvent *)event {
    if (!self.isEnabled) return;


    NSSet *touches = [event touchesMatchingPhase:NSTouchPhaseTouching
inView:self.view];


    if (touches.count == 2) {
        self.initialPoint = [self.view convertPointFromBase:[event
locationInWindow]];
        NSArray *array = [touches allObjects];
        _initialTouches[0] = [[array objectAtIndex:0] retain];
        _initialTouches[1] = [[array objectAtIndex:1] retain];
        _currentTouches[0] = [_initialTouches[0] retain];
        _currentTouches[1] = [_initialTouches[1] retain];
    } else if (touches.count > 2) {
```

```
        // More than 2 touches. Only track 2.
        if (self.isTracking) {
            [self cancelTracking];
        } else {
            [self releaseTouches];
        }
    }
}
```

When one or both of the fingers move, the `touchesMovedWithEvent:` method is invoked. The view in `LightTable` implements this method as shown in Listing 8-6. Again, it gets all the touch objects associated with the event and, if there are exactly two touches, it matches the current touches with their initial counterparts. Then it computes the delta values for origin and size and, if these exceed a certain threshold, invokes an action method to perform the transformation.

Listing 8-6 Handling touches in `touchesMovedWithEvent:`

```
- (void)touchesMovedWithEvent:(NSEvent *)event {
    if (!self.isEnabled) return;
    self.modifiers = [event modifierFlags];
    NSSet *touches = [event touchesMatchingPhase:NSTouchPhaseTouching
inView:self.view];
    if (touches.count == 2 && _initialTouches[0]) {
        NSArray *array = [touches allObjects];
        [_currentTouches[0] release];
        [_currentTouches[1] release];

        NSTouch *touch;
        touch = [array objectAtIndex:0];
        if ([touch.identity isEqual:_initialTouches[0].identity]) {
            _currentTouches[0] = [touch retain];
        } else {
            _currentTouches[1] = [touch retain];
        }
        touch = [array objectAtIndex:1];
        if ([touch.identity isEqual:_initialTouches[0].identity]) {
```

```
            _currentTouches[0] = [touch retain];
        } else {
            _currentTouches[1] = [touch retain];
        }
        if (!self.isTracking) {
            NSPoint deltaOrigin = self.deltaOrigin;
            NSSize  deltaSize = self.deltaSize;
            if (fabs(deltaOrigin.x) > _threshold ||
                fabs(deltaOrigin.y) > _threshold ||
                fabs(deltaSize.width) > _threshold ||
                fabs(deltaSize.height) > _threshold) {
                self.isTracking = YES;
                if (self.beginTrackingAction)
                    [NSApp sendAction:self.beginTrackingAction to:self.view
from:self];
            }
        } else {
            if (self.updateTrackingAction)
              [NSApp sendAction:self.updateTrackingAction to:self.view from:self];
        }
    }
}
```

Listing 8-7 shows how the project computes the delta values for transforming the origin of the view. Note how the code uses values from the `NSTouch` properties `normalizedPosition` and `deviceSize` in this computation.

Listing 8-7 Obtaining a delta value using `normalizedPosition` and `deviceSize`

```
- (NSPoint)deltaOrigin {
    if (!(_initialTouches[0] && _initialTouches[1] &&
        _currentTouches[0] && _currentTouches[1])) return NSZeroPoint;


    CGFloat x1 = MIN(_initialTouches[0].normalizedPosition.x,
_initialTouches[1].normalizedPosition.x);
    CGFloat x2 = MAX(_currentTouches[0].normalizedPosition.x,
_currentTouches[1].normalizedPosition.x);
```

```
    CGFloat y1 = MIN(_initialTouches[0].normalizedPosition.y,
_initialTouches[1].normalizedPosition.y);
    CGFloat y2 = MAX(_currentTouches[0].normalizedPosition.y,
_currentTouches[1].normalizedPosition.y);

    NSSize deviceSize = _initialTouches[0].deviceSize;
    NSPoint delta;
    delta.x = (x2 - x1) * deviceSize.width;
    delta.y = (y2 - y1) * deviceSize.height;
    return delta;
}
```

Finally, the view implements the `touchesEndedWithEvent:` and `touchesCancelledWithEvent:` methods, as shown in Listing 8-8, primarily to cancel tracking of the event.

Listing 8-8 Handling ending and cancelled touches

```
- (void)touchesEndedWithEvent:(NSEvent *)event {
    if (!self.isEnabled) return;
    self.modifiers = [event modifierFlags];
    [self cancelTracking];
}

- (void)touchesCancelledWithEvent:(NSEvent *)event {
    [self cancelTracking];
}
```

Para

Mouse Events and the Trackpad

The operating system interprets a single finger moving across the trackpad as a mouse-tracking event, and moves the mouse pointer in a corresponding speed and direction. The single-finger movement generates both gesture events and mouse events, although the gesture events are ignored unless the view accept touches.

Movements of more than one finger on the trackpad do not move the cursor, although they may still result in the generation of mouse events used in scrolling. An exception to this is a buttonless trackpad, where two fingers may be physically touching the trackpad while the cursor is moving because one of them is resting.

If a mouse event is generated by a touch event, the subtype of the mouse event is `NSTouchEventSubtype`. You could evaluate mouse-event objects to determine when to ignore mouse events in favor of touch events when they both are generated. (This advice is not applicable to scroll wheel events.)

You should not attempt to extract touches from a mouse event using the `touchesMatchingPhase:inView:` method. Although you can check the subtype of the event object to see if a touch generated the mouse event, you can not correlate the mouse event to any particular touch. Further, you cannot query a touch event to find out if it also generated a mouse event.

Monitoring Events

The AppKit framework allows you to install an event monitor, an object that looks for user-input events of a certain type (or types) as an application dispatches them in its `sendEvent:` method. For example, a monitor could look for mouse-up, key-down, or swipe-gesture events, or even all events.

There are two kinds of event monitors, each differing in monitoring scope and capabilities:

- A **global event monitor** looks for user-input events dispatched to applications other than the one in which it is installed. The monitor cannot modify an event or prevent its normal delivery. And it may only monitor key events if accessibility is enabled or if the application is trusted for accessibility.

  You install a global event monitor with the `NSEvent` class method `addGlobalMonitorForEventsMatchingMask:handler:`.

- A **local event monitor** looks at user-input events that are being dispatched to the application in which the monitor is installed. For a given event object of interest, the local monitor can return the object unmodified, create and return a new `NSEvent` object, or return `nil` to stop the dispatching of the event.

  You install a local event monitor with the `NSEvent` class method `addLocalMonitorForEventsMatchingMask:handler:`.

  Note: Local and global event monitors are mutually exclusive. For example, the global monitor does not observe the event stream of the application in which it is installed. The local event monitor only observes the event stream of its application. To monitor events from all applications, including the "current" application, you must install both event monitors.

The parameters of both monitor-installation methods are nearly identical. The first parameter is an event mask for specifying the events of interest by type. The second parameter defines a block that performs the handling of monitored events; it is called for each new event that matches one of the specified types. For both methods, an `NSEvent` object is the sole argument of the block. However, the block handler for `addLocalMonitorForEventsMatchingMask:handler:` is typed to return an `NSEvent` object while the block handler for the global method returns `void`. The handlers are always called on the main thread. Both class methods return the monitor object, which the calling object does not own (and thus has no need to retain or release).

Important: After you finish monitoring events, you should remove the monitor object with a call to the `NSEvent` class method `removeMonitor:`. Garbage-collected applications should never remove an event monitor in the `finalize` method. Although memory-managed applications can remove the monitor in the `dealloc` method, that is discouraged. In both cases, removal of the event monitor should occur much earlier in the application life cycle.

There are many scenarios where an event monitor might be useful to an application. One example is a pop-up window that acts like a menu. The application wants to know when the user clicks outside of that window so it can dismiss it. It also wants to know if the user presses the Escape key (to dismiss it without saving changes) or if the user presses the Enter key (to dismiss it and save changes). The *AnimatedTableView* sample code project installs a local event monitor (in `ATColorTableController.m`) that performs these functions. Listing 9-1 shows how it does this.

Listing 9-1 Installing a local event monitor

```
- (void)editColor:(NSColor *)color locatedAtScreenRect:(NSRect)rect {

    // code unrelated to event monitoring deleted here.....

    // Start watching events to figure out when to close the window
    NSAssert(_eventMonitor == nil, @"_eventMonitor should not be created yet");
    _eventMonitor = [NSEvent addLocalMonitorForEventsMatchingMask:
            (NSLeftMouseDownMask | NSRightMouseDownMask | NSOtherMouseDownMask |
NSKeyDownMask)
            handler:^(NSEvent *incomingEvent) {
        NSEvent *result = incomingEvent;
        NSWindow *targetWindowForEvent = [incomingEvent window];
        if (targetWindowForEvent != _window) {
            [self _closeAndSendAction:NO];
        } else if ([incomingEvent type] == NSKeyDown) {
            if ([incomingEvent keyCode] == 53) {
                // Escape
                [self _closeAndSendAction:NO];
                result = nil; // Don't process the event
            } else if ([incomingEvent keyCode] == 36) {
                // Enter
                [self _closeAndSendAction:YES];
```

```
                result = nil;
            }
        }
        return result;
    }];
}
```

When the window is closed, the application has no more need for the event monitor. So it posts a notification when it closes the window. The method in Listing 9-2 is invoked as a result of this notification, and the class implements it to remove the event monitor (among other things).

Listing 9-2 Removing an event monitor

```
- (void)_windowClosed:(NSNotification *)note {
    if (_eventMonitor) {
        [NSEvent removeMonitor:_eventMonitor];
        _eventMonitor = nil;
    }
    [[NSNotificationCenter defaultCenter] removeObserver:self
name:NSWindowWillCloseNotification object:_window];
    [[NSNotificationCenter defaultCenter] removeObserver:self
name:NSApplicationDidResignActiveNotification object:nil];
}
```

Although event monitoring can be the ideal solution for some problems, it might not be the best for other ones. For example, the *AnimatedTableView* application installs a local event monitor, which can detect mouse events sent to the application but cannot detect mouse events sent to other applications. But the application needs to dismiss the window if the user clicks in another application. To do this, *AnimatedTableView* observes the `NSApplicationDidResignActiveNotification` notification instead of installing a global event monitor. A global event monitor would not be able to detect Command-Tab or a system alert, both of which should cause the window to be dismissed. Event monitors should be used only when there is no other way to solve your problem.

Text System Defaults and Key Bindings

This document reveals some tips and tricks about various defaults you can use to customize the behavior of Cocoa's text system. It also describes how to customize the key bindings supported by the text system.

Heavy-duty subclassers may alter some or all of the text system's functionality, rendering some or all of these features inactive.

Key Bindings

The text system uses a generalized key-binding mechanism that is completely re-mappable by the user, although defining custom key bindings dynamically (that is, while the application is running) is not supported. Key bindings are specified in an dictionary file that must have an extension of `.dict`; the format of this file should be an XML property list, but the text system can also understand old-style (NeXT era) property lists. The standard key bindings are specified in `/System/Library/Frameworks/AppKit.framework/Resources/StandardKeyBinding.dict`. These standard bindings include a large number of Emacs-compatible control key bindings, all the various arrow key bindings, bindings for making field editors and some keyboard UI work, and backstop bindings for many function keys.

To customize bindings, you create a file named `DefaultKeyBinding.dict` in `~/Library/KeyBindings/` and specify bindings to augment or replace the standard bindings. You may use the standard bindings file as a template. It is recommended that you use the Property List Editor application to edit a bindings dictionary. You may use another application such as TextEdit or Xcode, but if you do you must ensure the encoding of the saved file is UTF8.

Key bindings are key-value pairs with the key being a string that specifies a physical key and the value identifying an action method to be invoked when the key is pressed. (Many of these action methods are declared by `NSResponder`.) You can compose physical-key strings using the following elements:

- The alphanumeric character that appears on a US keyboard. For example, "f" or ">". (As noted below, some special characters are reserved for modifier flags.)
- For a few keys, such as Escape, Tab, and backward Delete (BS), the octal number from the ASCII table that identifies the key. For example, the octal number identifying the Escape key (sometimes used as a modifier key) is `\033`.

- An `enum` constant assigned a unique Unicode value that is used to identify a function key. These constants are defined in `NSEvent.h`. Examples of these constants are `NSF7FunctionKey`, `NSHomeFunctionKey`, and `NSHelpFunctionKey`.
- One or more key modifiers, which must precede one of the other key-identifier elements. The following special characters are used for modifier flags:
  - "^" for Control
  - "~" for Option
  - "$" for Shift
  - "#" for numeric keypad

  For example, the following string would identify the 0 (zero) key on the numeric keypad when the Control key is pressed simultaneously: "`^#0`".

The text system supports the specification of multiple keystroke bindings through nested binding dictionaries. For instance, Escape could be bound to `cancel:` or it could be bound to a whole dictionary which would then contain bindings for the next keystroke after Escape.

The following sample binding files illustrate how you might customize bindings. The first one adds Option-key bindings for some common Emacs behavior. This might be useful where the Option key bindings are not standard. With these bindings it would be necessary to type "`Control-Q, Option-f`" in order to type a florin character instead of moving forward a word. This sample also explicitly binds Escape to `complete:`. (In OS X, this is the default so this override changes nothing.)

```
/* ~/Library/KeyBindings/DefaultKeyBinding.dict */

{
    /* Additional Emacs bindings */
    "~f" = "moveWordForward:";
    "~b" = "moveWordBackward:";
    "~<" = "moveToBeginningOfDocument:";
    "~>" = "moveToEndOfDocument:";
    "~v" = "pageUp:";
    "~d" = "deleteWordForward:";
    "~^h" = "deleteWordBackward:";
    "~\010" = "deleteWordBackward:";  /* Option-backspace */
    "~\177" = "deleteWordBackward:";  /* Option-delete */
```

```
    /* Escape should really be complete: */
    "\033" = "complete:";  /* Escape */
}
```

The following example shows how to have multi-keystroke bindings. It binds a number of Emacs meta bindings using Escape as the meta key instead of the Option modifier. So Escape followed by the "f" key means `moveWordForward:` here. This sample binds Escape-Escape to `complete:`. Note the nested dictionaries

```
/* ~/Library/KeyBindings/DefaultKeyBinding.dict */
{
    /* Additional Emacs bindings */
    "\033" = {
        "\033" = "complete:";  /* ESC-ESC */
        "f" = "moveWordForward:";  /* ESC-f */
        "b" = "moveWordBackward:";  /* ESC-b */
        "<" = "moveToBeginningOfDocument:";  /* ESC-< */
        ">" = "moveToEndOfDocument:";  /* ESC-> */
        "v" = "pageUp:";  /* ESC-v */
        "d" = "deleteWordForward:";  /* ESC-d */
        "^h" = "deleteWordBackward:";  /* ESC-Ctrl-H */
        "\010" = "deleteWordBackward:";  /* ESC-backspace */
        "\177" = "deleteWordBackward:";  /* ESC-delete */
    };
}
```

Once you have completed specifying key bindings, you must save the file and relaunch the application for the bindings to take effect. With the right combination of key bindings and default settings, it should be possible to tailor the text system to your preferences.

Standard Action Methods for Selecting and Editing

The `NSResponder` class declares method prototypes for a number of standard action methods, nearly all related to manipulating selections and editing text. These methods are typically invoked through `doCommandBySelector:` as a result of interpretation by the input manager. They fall into the following general groups:

- Selection movement and expansion
- Text insertion
- General deletion of elements
- Modifying selected text
- Scrolling a document

In most cases the intent of the action method is clear from its name. The individual method descriptions in this specification also provide detailed information about what such a method should normally do. However, a few general concepts apply to many of these methods, and are explained here.

Selection Direction

Some methods refer to spatial directions; left, right, up, down. These are meant to be taken literally, especially in text. To accommodate writing systems with directionality different from Latin script, the terms forward, beginning, backward, and end are used.

Selection and insertion point

Methods that refer to moving, deleting, or inserting imply that some elements in the responder are selected, or that there's a zero-length selection at some location (the insertion point). These two things must always be treated consistently. For example, the `insertText:` method is defined as replacing the selection with the text provided. The `moveForwardAndModifySelection:` method extends or contracts a selection, even if the selection is merely an insertion point. When a selection is modified for the first time, it must always be extended. So a `moveForward...` message extends the selection from its end, while a `moveBackward...` message extends it from its beginning.

Marks

A number of action methods for editing text imitate the Emacs concepts of point (the insertion point), and mark (an anchor for larger operations normally handled by selections in graphical interfaces). The `setMark:` method establishes the mark at the current selection, which then remains in effect until the mark is changed again. The `selectToMark:` method extends the selection to include the mark and all characters between the selection and the mark.

The kill buffer

Also like Emacs, deletion methods affecting lines, paragraphs, and the mark implicitly place the deleted text into a buffer, separate from the pasteboard, from which you can later retrieve it. Methods such as `deleteToBeginningOfLine:` add text to this buffer, and `yank:` replaces the selection with the item in the kill buffer.

Text System Defaults

NSMnemonicsWorkInText

Allowed value: "YES" or "NO".

This default controls whether the text system accepts key events with the Option key down. The default value is `NO`. A value of `YES` means that any key event with the Option bit on will be passed up the responder chain to eventually be treated as a mnemonic instead of being accepted by the text as textual input or a key binding command. If this default is set to NO then the key events with the Option bit set will be passed through the text system's normal key input sequence. This will allow any key bindings involving Option to work (such as Emacs-style bindings like Option-f for word forward) and it allows typing of special international and Symbol font characters.

NSRepeatCountBinding

Allowed value: Key-binding style string.

This default controls the numeric argument binding. The default is for numeric arguments not to be supported. If you provide a binding for this default you enable the feature. This allows you to repeat a keyboard command a given number of times. For instance "`Control-U 10 Control-F`" means move forward ten characters.

NSQuotedKeystrokeBinding

Allowed value: Key-binding style string.

This default controls the quote binding. The default is for this to be "^q" (that's Control-Q). This is the binding that allows you to literally enter characters that would otherwise be interpreted as commands. For instance "`Control-Q Control-F`" would insert a Control-F character into the document instead of performing the command `moveForward:`.

NSTextShowsInvisibleCharacters

Allowed value: "YES" or "NO".

The default controls whether a text object will by default show invisible characters like tab, space, and carriage return using some visible glyph. By default it is `NO`. It only controls the default setting for `NSLayoutManager` objects (which can be modified programmatically). In order for this to work, the rule book generating the glyphs must support the feature. Currently our rule books do not support this feature, so currently this default is not very useful.

NSTextShowsControlCharacters

Allowed value: "YES" or "NO".

The default controls whether a text object will by default show control characters visibly (usually by representing Control-C as "^C" in the text). By default it is `NO`. It only controls the default setting for `NSLayoutManager` objects (which can be modified programmatically). In order for this to work, the rule book(s) generating the glyphs must support the feature. This feature carries a cost. It will increase the memory needed for documents that contain control characters by quite a lot. Use it with care.

NSTextSelectionColor

Allowed value: Color object or specifier.

This default controls the background color of selected text. By default this is light gray. Defaults that accept colors accept them in one of three ways. Either as an archived `NSColor` object, or as three RGB components, or as a string that can be resolved to a factory selector on `NSColor` that will return the desired color (for example, "redColor"). Note that NSTextField objects and other controls that use field editors to edit their text control their own selection attributes to conform with the UI.

NSMarkedTextAttribute and NSMarkedTextColor

Allowed value: Color object/specifier or "underline".

This default controls the way that marked text is displayed. The `NSMarkedTextAttribute` can be either "`Background`" or "`Underline`". If it is "`Background`" then `NSMarkedTextColor` indicates the background color to use for marked text. If `NSMarkedTextAttribute` is "`Underline`", `NSMarkedTextColor` indicates the foreground color to use for marked text (the marked text will be drawn in the indicated color and underlined). By default, marked text is drawn with a yellowish background color. Kit defaults that accept colors accept them in one of three ways. Either as an archived `NSColor` object, or as three RGB components, or as a string that can be resolved to a factory selector on `NSColor` that will return the desired color (for example, "redColor"). If

the `NSMarkedTextAttribute` default contains a color instead of one of the strings "`Background`" or "`Underline`" then that color is used as the background color for marked text and the `NSMarkedTextColor` attribute is ignored.

NSTextKillRingSize

Allowed value: Number string.

This default controls the size of the kill ring (as in Emacs Control-Y). The default value is 1 (not really a ring at all, just a single buffer). If you set this to a value larger than one, you also need to rebind Control-Y to `yankAndSelect:` instead of `yank:` for things to work properly (note that `yankAndSelect:` is not listed in any headers). See "Key Bindings" (page 105) for more information about bindings.

Mouse-Tracking and Cursor-Update Events

Mouse-tracking messages are sent to an object when the mouse pointer (without a mouse button being pressed) enters and exits a region of a window. This region is known as a tracking rectangle or tracking area. Mouse tracking enables the view owning the region to respond, for example, by drawing a highlight color or displaying a tool tip. Cursor-update events are a special kind of mouse-tracking event that the Application Kit handles automatically. When the mouse pointer enters a cursor rectangle, the Application Kit displays a cursor image appropriate to the type of view under the rectangle; for example, when a mouse pointer enters a text view, an I-beam cursor is displayed instead.

The sections in this chapter describe how you set up tracking rectangles and respond to mouse-tracking events. They also discuss how to specify and manage the rectangles for cursor-update events.

Important: This appendix describes a legacy API for handling mouse-tracking and cursor-update events. The preferred API for these tasks is the API defined by the `NSTrackingArea` class and related methods declared in the `NSView` class. See "Using Tracking-Area Objects" (page 78) for details.

Handling Mouse-Tracking Events

A region of a view set up for tracking mouse movement is known as a tracking rectangle. When the mouse cursor enters the tracking rectangle, the Application Kit sends mouse-entered events (type `NSMouseEntered`) to the object owning the rectangle (which is not necessarily the view itself); when the cursor leaves the rectangle, the Application Kit sends the object mouse-exited events (type `NSMouseExited`). These events correspond to the `mouseEntered:` and `mouseExited:` methods, respectively, of `NSResponder`. Mouse tracking can be useful for such tasks as displaying context-sensitive messages or highlighting graphic elements under the cursor. An `NSView` object can have any number of tracking rectangles, which can overlap or be nested one within the other; the `NSEvent` objects generated to represent mouse-tracking events include a tag (accessed through the `trackingNumber` method) that identifies the rectangle associated with an event.

Important: The proper order of mouse-entered and mouse-exited events received by tracking rectangles in an application cannot be guaranteed. For example, if you move the mouse cursor from one tracking rectangle to another tracking rectangle in a view and back, the order of events (as messages) could be: `mouseEntered:`, `mouseEntered:`, `mouseExited:`, `mouseExited:`.

To create a tracking rectangle, send a `addTrackingRect:owner:userData:assumeInside:` message to the `NSView` object associated with the rectangle, as shown in "Managing a Tracking-Area Object." This method registers an owner for the tracking rectangle, so that the owner receives the tracking-event messages. The owner is typically the view object itself, but need not be. The method returns the tracking rectangle's tag so that you can store it for later reference in the event-handling methods `mouseEntered:` and `mouseExited:`. To remove a tracking rectangle, use the `removeTrackingRect:` method, which takes as an argument the tag of the tracking rectangle to remove.

Listing A-1 Adding a tracking rectangle to a view region

```
- (void)viewDidMoveToWindow {
    // trackingRect is an NSTrackingRectTag instance variable
    // eyeBox is a region of the view (instance variable)
    trackingRect = [self addTrackingRect:eyeBox owner:self userData:NULL
assumeInside:NO];
}
```

In the above example, the custom view adds the tracking rectangle in the `viewDidMoveToWindow` method instead of `initWithFrame:`. Although `NSView` implements the `addTrackingRect:owner:userData:assumeInside:` method, a view's window maintains the list of tracking rectangles. When a view's `initWithFrame:` initializer is invoked, the view is not yet associated with a window, so the tracking rectangle cannot yet be added to the window's list. Thus the best place to add tracking rectangles initially is in the `viewDidMoveToWindow` method.

Tracking rectangle bounds are inclusive for the top and left edges, but not for the bottom and right edges. Thus, if you have a unflipped view with a tracking rectangle covering its bounds, and the view's frame has the geometry `frame.origin = (100, 100), frame.size = (200, 200)`, then the area for which the tracking rectangle is active is `frame.origin = (100, 101), frame.size = (199, 199)`, in frame coordinates.

Tracking rectangles can also be used to provide `NSMouseMoved` events to views in the `mouseMoved:` method. For a view to receive `NSMouseMoved` events, however, two things must happen:

- The view must be the first responder.
- The view's window must be sent a `setAcceptsMouseMovedEvents:` message with an argument of `YES`.

As noted in "Event Objects and Types" (page 27) and "Handling Mouse Events" (page 51), an `NSWindow` object by default does not receive `NSMouseMoved` events because they can easily flood the event queue. If you only want to receive mouse-moved messages while the mouse is over your view, you should turn them off again when a mouse-tracking session completes.

You typically send the `setAcceptsMouseMovedEvents:` message (with an argument of `YES`) in your implementation of `mouseEntered:`. If you want to turn them off after a tracking session ends, you can send the message again with an argument of `NO` in your implementation of `mouseExited:`. However, you should also set the window state back to what it was before you turned on mouse-moved events to ensure that the window does not stop receiving mouse-moved events if it wants them for other purposes.

The tracking code in Listing 6-2 is used in making an "eyeball" follow the movement of the mouse pointer when it enters a tracking rectangle. Note that the `mouseEntered:` implementation uses the `wasAcceptingMouseEvents` instance variable to capture the window's current state as regards mouse-moved events before these events are turned on for the current tracking session; later, in `mouseExited:`, the value of this instance variable is used as the argument to `setAcceptsMouseMovedEvents:`, thereby resetting window state.

Listing A-2 Handling mouse-entered, mouse-moved, and mouse-exited events

```
- (void)mouseEntered:(NSEvent *)theEvent {
    wasAcceptingMouseEvents = [[self window] acceptsMouseMovedEvents];
    [[self window] setAcceptsMouseMovedEvents:YES];
    [[self window] makeFirstResponder:self];
   NSPoint eyeCenter = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    eyeBox = NSMakeRect((eyeCenter.x-10.0), (eyeCenter.y-10.0), 20.0, 20.0);
    [self setNeedsDisplayInRect:eyeBox];
    [self displayIfNeeded];
}

- (void)mouseMoved:(NSEvent *)theEvent {
   NSPoint eyeCenter = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    eyeBox = NSMakeRect((eyeCenter.x-10.0), (eyeCenter.y-10.0), 20.0, 20.0);
    [self setNeedsDisplayInRect:eyeBox];
    [self displayIfNeeded];
}

- (void)mouseExited:(NSEvent *)theEvent {
```

```
    [[self window] setAcceptsMouseMovedEvents:wasAcceptingMouseEvents];
    [self resetEye];
    [self setNeedsDisplayInRect:eyeBox];
    [self displayIfNeeded];
}
```

Because tracking rectangles are maintained by `NSWindow` objects, a tracking rectangle is a static entity; it doesn't move or change its size when an `NSView` object does. If you use tracking rectangles, you should be sure to remove and reestablish them when you change the frame rectangle of the view object that contains them. If you're creating a custom subclass of `NSView`, you can override the `setFrame:` and `setBounds:` methods to do this, as shown in "Compatibility Issues." If your class is not a custom view class, you can register your class instance as an observer for the notification `NSViewFrameDidChangeNotification` and have it reestablish the tracking rectangles on receiving the notification.

Listing A-3 Resetting a tracking rectangle

```
- (void)setFrame:(NSRect)frame {
    [super setFrame:frame];
    [self removeTrackingRect:trackingRect];
    [self resetEye];
    trackingRect = [self addTrackingRect:eyeBox owner:self userData:NULL
assumeInside:NO];
}

- (void)setBounds:(NSRect)bounds {
    [super setBounds:bounds];
    [self removeTrackingRect:trackingRect];
    [self resetEye];
    trackingRect = [self addTrackingRect:eyeBox owner:self userData:NULL
assumeInside:NO];
}
```

You should also remove the tracking rectangle when your view is removed from its window, which can happen either because the view is moved to a different window, or because the view is removed as part of deallocation. One place to do this is the `viewWillMoveToWindow:` method, as shown in "Compatibility Issues."

Listing A-4 Removing a tracking rectangle when a view is removed from its window

```
- (void)viewWillMoveToWindow:(NSWindow *)newWindow {
    if ( [self window] && trackingRect ) {
        [self removeTrackingRect:trackingRect];
    }
}
```

Managing Cursor-Update Events

One common use of tracking rectangles is to change the cursor image over different types of graphic elements. Text, for example, typically requires an I-beam cursor. Changing the cursor is such a common operation that `NSView` defines several convenience methods to ease the process. A tracking rectangle generated by these methods is called a cursor rectangle. The Application Kit itself assumes ownership of cursor rectangles, so that when the user moves the mouse over the rectangle the cursor automatically changes to the appropriate image. Unlike general tracking rectangles, cursor rectangles may not partially overlap. They may, however, be completely nested, one within the other.

Because cursor rectangles need to be reset often as a view's size and graphic elements change, `NSView` defines a single method, `resetCursorRects`, that's invoked any time its cursor rectangles need to be reestablished. A concrete subclass overrides this method, invoking `addCursorRect:cursor:` for each cursor rectangle it wishes to set (as illustrated in Listing A-5). Thereafter, the view's cursor rectangles can be rebuilt by invoking the `NSWindow` method `invalidateCursorRectsForView:`. Before `resetCursorRects` is invoked, the owning view is automatically sent a `disableCursorRects` message to remove existing cursor rectangles.

Listing 6-4 shows an implementation of `resetCursorRects`.

Listing A-5 Resetting a cursor rectangle

```
-(void)resetCursorRects
{
    [self addCursorRect:[self calculatedItemBounds] cursor:[NSCursor
openHandCursor]];


}
```

Although you can temporarily remove a single cursor rectangle with `removeCursorRect:cursor:`, you should rarely need to do so. Whenever cursor rectangles need to be rebuilt, `NSView` invokes `resetCursorRects` so that you can establish only the cursor rectangles needed. If you implement `resetCursorRects` in this way, you can then simply modify the state this method uses to build its cursor rectangles and then invoke the `NSWindow` method `invalidateCursorRectsForView:`.

An `NSView` object's cursor rectangles are automatically reset whenever:

- Its frame or bounds rectangle changes, whether by a `setFrame...` or `setBounds...` message or by autoresizing.
- Its window is resized. In this case all of the window's view objects get their cursor rectangles reset.
- It is moved in the view hierarchy.
- It is scrolled in an `NSScrollView` or `NSClipView` object.

You can temporarily disable all the cursor rectangles in a window using the `NSWindow` method `disableCursorRects` and enable them again with the `enableCursorRects` method. The `areCursorRectsEnabled` method of `NSWindow` tells you whether they're currently enabled.

Document Revision History

This table describes the changes to *Cocoa Event Handling Guide*.

Date	Notes
2013-01-28	Removed obsolete information on Xcode key bindings; see Xcode's documentation for information on Xcode key bindings. See "Key Bindings Preferences" in *Xcode Workspace Guide* for details of using key bindings in Xcode.
2012-05-14	Updated links to other documents.
2010-05-20	Updated paths to examples in See Also. All examples are now in ADC Reference Library.
2009-10-19	Added "Handling Trackpad Events" and "Monitoring Events" chapters, which document features introduced in OS X v10.6. Also added information on NSEvent class methods introduced in OS X v10.6.
2009-08-28	Corrected ordering of NSWindowController in the responder chain. Added links to Cocoa Core Competencies. Made minor corrections.
2009-02-04	Added OS X v10.5 details about scroll-wheel events and key equivalents.
2008-11-19	Added section on handling non-mouse events during tracking operations. Made an assortment of small corrections.
2007-03-16	Added chapter on using NSTrackingArea objects (new in OS X v10.5). Clarified behavior of cursor-rect methods. Added note to "Text System Defaults and Key Bindings" stating that defining custom key bindings dynamically is not supported
2006-07-24	Made substantial changes as part of a major update and reorganization of Cocoa documents related to event handling.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010