NSResponder

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: AppKit/NSResponder.h

Class Description

NSResponder is an abstract class that forms the basis of event and command processing in the Application Kit. The core classes—NSApplication, NSWindow, and NSView—inherit from NSResponder, as must any class that handles events. The responder model is built around three components: event messages, action messages, and the responder chain. An *event message* is a message corresponding directly to an input event, and includes as its sole argument an NSEvent object describing the event; a mouse down or keypress, for example. An *action message* is a higher-level message indicating a command to be performed, which includes as an argument the object requesting the action. Some examples of action messages are the standard **cut:**, **copy:**, and **paste:**.

The *responder chain* is a 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 message is passed to its successor in the chain. This allows responder objects to delegate responsibility to other, typically higher-level objects. The responder chain is constructed automatically as described below, but you can insert custom objects into it using the **setNextResponder:** method and examine it with **nextResponder**.

An application can contain any number of responder chains, but only one is active at any given time. It begins with the *first responder* in some NSWindow and proceeds to the NSWindow itself. The first responder is typically the "selected" NSView within the NSWindow, and its next responder is its containing NSView (also called its superview), and so on up to the NSWindow itself. You can safely inject other responders between NSViews, but you can't add responders past the NSWindow. Nearly all event messages apply to a single window's responder chain.

For action messages, a more elaborate responder chain is used, constructed from the individual responder chains of two NSWindows and the application object itself. The NSWindows are the *key window*, whose responder chain gets first crack at action messages, and the *main window*, which follows. The main window is sometimes identical to the key window; the two are typically distinguished when an auxiliary window or panel related to a primary window—such as a Find Panel—is opened. In this case the primary window, which was the key window, becomes the main window, and the Find Panel becomes key. The two windows and the NSApplication object also 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 (a **nextResponder** message

to one of these objects doesn't return the delegate). Given all these components, then, the full responder chain comprises these objects:

- The key window's first responder and successors, including objects added with **setNextResponder:**
- The key window itself
- The key window's delegate (which need not inherit from NSResponder)
- The main window's first responder and successors, including objects added with **setNextResponder:**
- The main window itself
- The main window's delegate (which need not inherit from NSResponder)
- The application object, NSApp
- The application object's delegate (which need not inherit from NSResponder)

Selecting the First Responder

The first responder is typically chosen by the user, with the mouse or keyboard. The mechanism by which one object loses its first responder status and another gains it is public though, and you can programmatically change the first responder if necessary. The method that changes the first responder is NSWindow's **makeFirstResponder:**. An NSWindow's first responder is initially itself, though you can set which object will be first responder when the NSWindow is first placed on-screen using the **setInitialFirstResponder:** method.

makeFirstResponder: always asks the current first responder if its ready to resign its status, using **resignFirstResponder**. If the current first responder returns NO when sent this message, **makeFirstResponder:** fails and likewise returns NO. If the current first responder returns YES then the new one is sent a **becomeFirstResponder** message to inform it that it can be the first responder. This object can return NO to reject the assignment, in which case the NSWindow itself becomes the first responder.

When an NSWindow that's the key window receives a mouse-down event, it automatically tries to make first responder the NSView under the event. It does so by asking the NSView whether it wants to become first responder, using the **acceptsFirstResponder** method defined by this class, with the mouse-down event as the argument. This method normally returns NO; responder subclasses that need to be first responder must override it to return YES. This method is also used when the user changes the first responder using the keyboard.

Normally a mouse-down event in a non-key window simply brings the window forward and makes it key, and isn't sent to the NSView over which it occurs. The NSView can claim an initial mouse-down, however, by implementing **acceptsFirstMouse:** to return YES. The argument is the mouse-down event, which the NSView can examine to determine whether it wants to receive the mouse event and potentially become first responder.

An additional consideration for responders that manage selections is of course to set the selection. An NSView that handles mouse events should set this itself. However, objects can also define methods for setting their selection that automatically make the receiver first responder as well. NSTextField's **selectText:**, for example, does something quite like this.

Event and Action Messages in the Responder Chain

The main purpose of the responder chain is to route events and action messages to an appropriate target. Event and action methods are dispatched in different ways, by different methods. Nearly all events enter an application from the Window Server, and are handled automatically by NSApplication's **sendEvent:** method. Action messages are instigated by objects, who use NSApplication's **sendAction:to:from:** method to route them to their proper destinations.

NSApplication's **sendEvent:** analyzes the event and handles some things specially—key equivalents, for example. Most events, however, it passes to the appropriate window for dispatch up its responder chain using NSWindow's **sendEvent:** method. NSResponder's default implementations of all event methods simply pass the message to the next responder, so if no object in the responder chain does anything with the event it's simply lost. As mentioned before, an NSView's next responder is nearly always its superview, so if, for example, the NSView that receives a **mouseDown:** message doesn't handle it, its superview gets a chance, and so on up to the NSWindow. 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.

Event messages form a well-known set, so NSResponder provides implementations for all of them. Action messages, however, are defined by custom classes and can't be predicted. For this reason they're dispatched in different manner from events. To instigate an action message, an object invokes NSApplication's **sendAction:to:from:**. The first argument is the selector for the action method to invoke. The second is the intended recipient of the message, often called the *target*. The final argument is usually the object invoking **sendAction:to:from:**, thus indicating which object instigated the action message. 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*, where the target is **nil**, **sendAction:to:from:** searches the full responder chain for an object that implements the action method specified. If it finds one, it sends the message to that object with the instigator of the action message as the sole argument. The receiver of the action message can then use the argument directly as input or query it for additional information. You can find the recipient of an untargeted action message without actually sending the message using **targetForAction:**.

A more general mechanism, which applies to the shorter form of the responder chain, is provided by NSResponder's **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 NSApplication's **sendAction:to:from:** 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 beeps.

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

Implementing Event and Action Methods

Implementing event methods is fairly straightforward. If your subclass handles a particular event, it overrides the method—**keyDown:**, for example—usurping the implementation of its superclass. 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 **super**'s 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. "Key Events," below, describes how to handle keyboard events in your application. See the NSView class specification for information on handling mouse events.

Action methods don't have default implementations, so responder subclasses shouldn't blindly forward action messages to **super**. Passing of action messages 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.

Key Events

Processing keyboard input is by far the most complex part of event handling. 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. The sections below attempt to explain how events are handled through the operating system and the Application Kit, so that you can understand what your objects receive and don't receive.

The Path of a Key Event

Physical keyboard events must pass through the operating system before becoming NSEvent objects in the Application Kit. Depending on the operating system, some of these "raw" events might be trapped before that ever happens. Reserved key combinations are often trapped in this way. Key events that arrive at the Application Kit are processed by NSApplication's **sendEvent:** method as indicated before. The application object filters out key equivalents (also known as "Command key events") and sends them out as described below under "Key Equivalents and Mnemonics." All other key events are passed to the key window's **sendEvent:** method.

The key window first checks the event to see if the Control key is pressed. If it is, the window treats the event as a forced control event, which is blocked from the responder chain and is processed immediately as a potential mnemonic or keyboard interface control event. If this doesn't apply, the event is passed to the window's first responder in a **keyDown:** message, which is how your custom responders receive uninterpreted key events. "Keyboard Input" describes how you can handle these events.

If no view object in the key window accepts the key event, NSWindow's **keyDown:** attempts to handle the key event itself. It tries to interpret the key event as each of the following, in order, beeping if it fails to match any of them to let the user know that the typing couldn't be processed:

- A mnemonic matching the character(s) typed, not requiring the Alternate key to be pressed

- A key equivalent, not requiring the Command (or Control) key to be pressed
- A keyboard interface control event

Key Equivalents and Mnemonics

A key equivalent is a character bound to some view in a window, which causes that view to perform a specified action when the user types that character, usually while pressing the Command key (the Control key on Microsoft Windows). A mnemonic works similarly, using the Alternate key as its cue to action. If both modifier keys are pressed, the key event is interpreted only as a mnemonic. A key equivalent or mnemonic must be a character that can be typed with no modifier keys, or with Shift only. Each is sent down the view hierarchy of a window instead of up the responder chain, but at different times.

Key equivalents are dispatched by the NSApplication object's **sendEvent:** method. On the Mach operating system, this results in a **performKeyEquivalent:** message being sent to every NSWindow in the application until one of them returns YES. On the Microsoft Windows operating system, it results in a **performKeyEquivalent:** message being sent to the menu of the key window, and of the main window if the key window's menu doesn't handle it. This difference in handling means that, among other things, NSWindow subclasses shouldn't override **performKeyEquivalent:**. Also, objects other than menu items shouldn't be assigned key equivalents; they should instead be assigned mnemonics. Key equivalents sent to a window on Mach are passed down the view hierarchy through NSView's abstract implementation of **performKeyEquivalent:**, which forwards the message to each of its subviews until one responds YES, returning NO if none does.

Mnemonics, on the other hand, are dispatched by the key window. If the user presses the Control key as well as the mnemonic's key combination, NSWindow's **sendEvent:** immediately treats that event as a mnemonic to be performed, without sending the event up the responder chain. If the user doesn't press the Control key, the event passes through the window's responder chain, possibly being handled by a responder, before arriving as a **keyDown:** message to the window. In either case, a mnemonic for a top-level menu on Microsoft Windows is sent back to the operating system, and eventually results in the Application Kit invoking a menu item's action. Any other mnemonic is handled by sending a **performMnemonic:** message down the window's view hierarchy, in the same manner as for a **performKeyEquivalent:** message.

Note: **performKeyEquivalent:** takes an NSEvent as its argument, while **performMnemonic:** takes an NSString containing the uninterpreted characters of the key event. You should extract the characters for a key equivalent using NSEvent's **charactersIgnoringModifiers** method.

Keyboard Interface Control

Mnemonics are actually part of a more general means of controlling the user interface via the keyboard. An NSWindow treats certain key events specially, as commands to move control to a different interface object, to simulate a mouse click on it, and so on. In brief, pressing Tab moves control to the next object, whether a button, a text field, or some other kind of control object. Shift-Tab moves control to the previous one. Pressing Space simulates a mouse click for many kinds of control objects, causing a push button to click, a radio button to toggle its state, and so on. In selection lists, pressing Space selects or deselects the

highlighted item; the user can also press Alternate or Shift to extend the selection, not affecting other selected items. Some interface controls also accept arrow-key input.

Each window can be assigned a default button, which is triggered by the Return or Enter key. Also, in modal windows or panels the user can press the Escape key to dismiss the window or panel. If interface control moves to another button, the default button temporarily loses this ability as the user's focus shifts to the button where control resides. However, if control then moves to a different kind of interface object, the default button resumes its normal ability.

The interface objects that are connected together within a window make up the window's *key view loop*. You normally set up the key view loop using Interface Builder, establishing connections to each interface object's **nextKeyView** outlet. You can also set the object that's initially selected when a window is first opened by setting the window's **initialFirstResponder** outlet in Interface Builder. NSView and NSWindow also define a number of methods for manipulating the key view loop programmatically; see their class specifications for more information.

Keyboard Input

A normal key event eventually makes its way to the responder chain as a **keyDown:** message, which the receiver can handle 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 to delete and move selected items. The receiver of a **keyDown:** message can extract the event's character's directly using NSEvent's **characters** or **charactersIgnoringModifiers** methods, or it can pass the key event to the Application Kit's input manager for interpretation according to the user's key bindings. Input management allows key events to be interpreted as text not directly available on the keyboard, such as Kanji and some accented characters, and as commands that affect the content of the responder object handling the event. See the NSInputManager and NSTextInput class and protocol specifications for more information on input management and key binding.

To invoke the input manager, simply invoke NSResponder's **interpretKeyEvents:** message in your implementation of **keyDown:**. This method sends an NSArray of events to the input manager, which interprets the events as text or commands and responds by sending **insertText:** or **doCommandBySelector:** to your responder object. The section below, "Standard Action Methods for Selecting and Editing," describes the messages that your object might get sent.

Standard Action Methods for Selecting and Editing

NSResponder declares 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. **moveForwardAndModifySelection:** 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). **setMark:** establishes the mark at the current selection, which then remains in effect until the mark is changed again. **selectToMark:** 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.

Other Uses

The responder chain is utilized by two other mechanisms in the Application Kit. In enabling and disabling a menu item, the application object consults the full responder chain for an object that implements the menu item's action method, as described in the NSMenuActionResponder protocol specification. 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. The Services validation process is described fully in "Services" in *OPENSTEP Programming Topics*.

Adopted Protocols

NSCoding	– encodeWithCoder: – initWithCoder:

Method Types

Changing the first responder	– acceptsFirstResponder – becomeFirstResponder – resignFirstResponder
Setting the next responder	– setNextResponder: – nextResponder
Event methods	– mouseDown: – mouseDragged: – mouseUp: – mouseMoved: – mouseEntered: – mouseExited: – rightMouseDown: – rightMouseDragged: – rightMouseUp: – keyDown: – keyUp: – flagsChanged: – helpRequested:
Special key event methods	– interpretKeyEvents: – performKeyEquivalent: – performMnemonic:
Clearing key events	– flushBufferedKeyEvents

Action methods	– capitalizeWord: – centerSelectionInVisibleArea: – changeCaseOfLetter: – complete: – deleteBackward: – deleteForward: – deleteToBeginningOfLine: – deleteToBeginningOfParagraph: – deleteToEndOfLine: – deleteToEndOfParagraph: – deleteToMark: – deleteWordBackward: – deleteWordForward: – indent: – insertBacktab: – insertNewLine: – insertNewlineIgnoringFieldEditor: – insertParagraphSeparator: – insertTab: – insertTabIgnoringFieldEditor: – insertText: – lowercaseWord: – moveBackward: – moveBackwardAndModifySelection: – moveDown: – moveDownAndModifySelection: – moveForward: – moveForwardAndModifySelection: – moveLeft: – moveRight: – moveToBeginningOfDocument: – moveToBeginningOfLine: – moveToBeginningOfParagraph: – moveToEndOfDocument: – moveToEndOfLine: – moveToEndOfParagraph: – moveUp: – moveUpAndModifySelection: – moveWordBackward: – moveWordBackwardAndModifySelection: – moveWordForward: – moveWordForwardAndModifySelection: – pageDown:

	– pageUp: – scrollLineDown: – scrollLineUp: – scrollPageDown: – scrollPageUp: – selectAll: – selectLine: – selectParagraph: – selectToMark: – selectWord: – setMark: – showContextHelp: – swapWithMark: – transpose: – transposeWords: – uppercaseWord: – yank:
Dispatch methods	– doCommandBySelector: – tryToPerform:with:
Terminating the responder chain	– noResponderFor:
Services menu updating	– validRequestorForSendType:returnType:
Setting the menu	– setMenu: – menu

Instance Methods

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

Overridden by subclasses to return YES if the receiver can handle key events and action messages sent up the responder chain. NSResponder's implementation returns NO, indicating that by default a responder object doesn't agree to become first responder. Objects that aren't first responder can receive mouse event messages, but no other event or action messages.

See also: – **becomeFirstResponder**, – **resignFirstResponder**, – **needsPanelToBecomeKey** (NSView)

becomeFirstResponder

– (BOOL)**becomeFirstResponder**

Notifies the receiver that it's about to become first responder in its NSWindow. NSResponder's implementation returns YES, accepting first responder status. Subclasses can override this method to update state or perform some action such as highlighting the selection, or to return NO, refusing first responder status.

Use NSWindow's **makeFirstResponder:**, not this method, to make an object the first responder. Never invoke this method directly.

See also: – **resignFirstResponder**, – **acceptsFirstResponder**

capitalizeWord:

– (void)**capitalizeWord:**(id)*sender*

Implemented by subclasses to capitalize the word or words surrounding the insertion point or selection, though without expanding the selection. If either end of the selection partially covers a word, that entire word is made lowercase. NSResponder declares, but doesn't implement this method.

See also: – **lowercaseWord:**, – **uppercaseWord:**, – **changeCaseOfLetter:**

centerSelectionInVisibleArea:

– (void)**centerSelectionInVisibleArea:**(id)*sender*

Implemented by subclasses to scroll the selection, whatever it is, inside its visible area. NSResponder declares, but doesn't implement this method.

See also: – **scrollLineDown:**, – **scrollLineUp:**, – **scrollPageDown:**, – **scrollPageUp:**

changeCaseOfLetter:

– (void)**changeCaseOfLetter:**(id)*sender*

Implemented by subclasses to change the case of a letter or letters in the selection, perhaps by opening a panel with capitalization options or by cycling through possible case combinations. NSResponder declares, but doesn't implement this method.

See also: – **lowercaseWord:**, – **uppercaseWord:**, – **capitalizeWord:**

complete:

– (void)**complete:**(id)*sender*

Implemented by subclasses to complete an operation in progress or a partially constructed element. This can be interpreted, for example, as a request to attempt expansion of a partial word, such as for expanding a glossary shortcut, or to close a graphic item being drawn. NSResponder declares, but doesn't implement this method.

deleteBackward:

– (void)**deleteBackward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single element backward from the insertion point (a letter or character in text, for example). NSResponder declares, but doesn't implement this method.

deleteForward:

– (void)**deleteForward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single element forward from the insertion point (a letter or character in text, for example). NSResponder declares, but doesn't implement this method.

deleteToBeginningOfLine:

– (void)**deleteToBeginningOfLine:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the beginning of a line (typically of text). Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **yank:**

deleteToBeginningOfParagraph:

– (void)**deleteToBeginningOfParagraph:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the beginning of a paragraph of text. Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **yank:**

deleteToEndOfLine:

– (void)**deleteToEndOfLine:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the end of a line (typically of text). Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

deleteToEndOfParagraph:

– (void)**deleteToEndOfParagraph:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all text from the insertion point to the end of a paragraph of text. Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **yank:**

deleteToMark:

– (void)**deleteToMark:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or all items from the insertion point to a previously placed mark, including the selection itself if not empty. Also places the deleted text into the kill buffer. NSResponder declares, but doesn't implement this method.

See also: – **setMark:**, – **selectToMark:**, – **yank:**

deleteWordBackward:

– (void)**deleteWordBackward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single word backward from the insertion point. NSResponder declares, but doesn't implement this method.

deleteWordForward:

– (void)**deleteWordForward:**(id)*sender*

Implemented by subclasses to delete the selection if there is one, or a single character or element forward from the insertion point. NSResponder declares, but doesn't implement this method.

doCommandBySelector:

– (void)**doCommandBySelector:**(SEL)*aSelector*

Attempts to perform the method indicated by *aSelector*. The method should take a single argument of type **id** and return **void**. If the receiver responds to *aSelector*, it invokes the method with **nil** as the argument. If the receiver doesn't respond, it sends this message to its next responder with the same selector. NSWindow and NSApplication also send the message to their delegates. If the receiver has no next responder or delegate, it beeps.

See also: – **tryToPerform:with:**, – **sendAction:to:from:** (NSApplication)

flagsChanged:

– (void)**flagsChanged:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed or released a modifier key (Shift, Control, and so on). NSResponder's implementation simply passes this message to the next responder.

flushBufferedKeyEvents

– (void)**flushBufferedKeyEvents**

Overridden by subclasses to clear any unprocessed key events.

helpRequested:

– (void)**helpRequested:**(NSEvent *)*theEvent*

Displays context-sensitive help for the receiver if such exists, otherwise passes this message to the next responder. NSWindow invokes this method automatically when the user clicks for help. Subclasses need not override this method, and application code shouldn't directly invoke it.

See also: – **showContextHelp:**

indent:

– (void)**indent:**(id)*sender*

Implemented by subclasses to indent the selection or the insertion point if there is no selection. NSResponder declares, but doesn't implement this method.

insertBacktab:

– (void)**insertBacktab:**(id)*sender*

Implemented by subclasses to handle a “backward tab.” A field editor might respond to this by selecting the field before it, while a regular text object either doesn’t respond to, or ignores such a message. NSResponder declares, but doesn’t implement this method.

insertNewline:

– (void)**insertNewline:**(id)*sender*

Implemented by subclasses to insert a line-break character at the insertion point or selection, deleting the selection if there is one, or to end editing if the receiver is a text field or other field editor. NSResponder declares, but doesn’t implement this method.

insertNewlineIgnoringFieldEditor:

– (void)**insertNewlineIgnoringFieldEditor:**(id)*sender*

Implemented by subclasses to insert a line-break character at the insertion point or selection, deleting the selection if there is one. Unlike **insertNewline:**, this method always inserts a line-break character and doesn’t cause the receiver to end editing. NSResponder declares, but doesn’t implement this method.

insertParagraphSeparator:

– (void)**insertParagraphSeparator:**(id)*sender*

Implemented by subclasses to insert a paragraph separator at the insertion point or selection, deleting the selection if there is one. NSResponder declares, but doesn’t implement this method.

insertTab:

– (void)**insertTab:**(id)*sender*

Implemented by subclasses to insert a tab character at the insertion point or selection, deleting the selection if there is one, or to end editing if the receiver is a text field or other field editor. NSResponder declares, but doesn’t implement this method.

insertTabIgnoringFieldEditor:

– (void)**insertTabIgnoringFieldEditor:**(id)*sender*

Implemented by subclasses to insert a tab character at the insertion point or selection, deleting the selection if there is one. Unlike **insertNewline:**, this method always inserts a tab character and doesn't cause the receiver to end editing. NSResponder declares, but doesn't implement this method.

insertText:

– (void)**insertText:**(NSString *)*aString*

Overridden by subclasses to insert *aString* at the insertion point or selection, deleting the selection if there is one. NSResponder's implementation simply passes this message to the next responder, or beeps if there is no next responder.

interpretKeyEvents:

– (void)**interpretKeyEvents:**(NSArray *)*eventArray*

Invoked by subclasses from their **keyDown:** method to handle a series of key events. This method sends the character input in *eventArray* to the system input manager for interpretation as text to insert or commands to perform. The input manager responds to the request by sending **insertText:** and **doCommandBySelector:** messages back to the invoker of this method. Subclasses shouldn't override this method.

See the NSInputManager and NSTextInput class and protocol specifications for more information on input management.

keyDown:

– (void)**keyDown:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed a key. The receiver can interpret *theEvent* itself, or pass it to the system input manager using **interpretKeyEvents:**. NSResponder's implementation simply passes this message to the next responder.

keyUp:

– (void)**keyUp:**(NSEvent *)*theEvent*

Informs the receiver that the user has released a key. NSResponder's implementation simply passes this message to the next responder.

lowercaseWord:

– (void)**lowercaseWord:**(id)*sender*

Implemented by subclasses to make lowercase every letter in the word or words surrounding the insertion point or selection, though without expanding the selection. If either end of the selection partially covers a word, that entire word is made lowercase. NSResponder declares, but doesn't implement this method.

See also: – **uppercaseWord:**, – **capitalizeWord:**, – **changeCaseOfLetter:**

menu

– (NSMenu *)**menu**

Returns the receiver's menu. For NSApplication this is the same as the menu returned by its **mainMenu** method.

See also: – **setMenu:**, – **menuForEvent:** (NSView), + **defaultMenu** (NSView)

mouseDown:

– (void)**mouseDown:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed the left mouse button. NSResponder's implementation simply passes this message to the next responder.

mouseDragged:

– (void)**mouseDragged:**(NSEvent *)*theEvent*

Informs the receiver that the user has moved the mouse with the left button pressed. NSResponder's implementation simply passes this message to the next responder.

mouseEntered:

– (void)**mouseEntered:**(NSEvent *)*theEvent*

Informs the receiver that the mouse has entered a tracking rectangle. NSResponder's implementation simply passes this message to the next responder.

mouseExited:

– (void)**mouseExited:**(NSEvent *)*theEvent*

Informs the receiver that the mouse has exited a tracking rectangle. NSResponder's implementation simply passes this message to the next responder.

mouseMoved:

– (void)**mouseMoved:**(NSEvent *)*theEvent*

Informs the receiver that the mouse has moved. NSResponder's implementation simply passes this message to the next responder.

See also: – **setAcceptsMouseMovedEvents:** (NSWindow)

mouseUp:

– (void)**mouseUp:**(NSEvent *)*theEvent*

Informs the receiver that the user has released the left mouse button. NSResponder's implementation simply passes this message to the next responder.

moveBackward:

– (void)**moveBackward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character backward. In text, if there is a selection it should be deselected, and the insertion point should be placed at the beginning of the former selection. NSResponder declares, but doesn't implement this method.

moveBackwardAndModifySelection:

– (void)**moveBackwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection backward by one element or character. If the end being modified is the backward end, this method expands the selection; if the end being modified is the forward end, it reduces the selection. The first **moveBackwardAndModifySelection:** or **moveForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the backward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveDown:

– (void)**moveDown:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character down. In text, if there is a selection it should be deselected, and the insertion point should be placed below the beginning of the former selection. NSResponder declares, but doesn't implement this method.

moveDownAndModifySelection:

– (void)**moveDownAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce the top or bottom end of the selection downward by one element, character, or line (whichever is appropriate for text direction). If the end being modified is the bottom, this method expands the selection; if the end being modified is the top, it reduces the selection. The first **moveDownAndModifySelection:** or **moveUpAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the bottom end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveForward:

– (void)**moveForward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character forward. In text, if there is a selection it should be deselected, and the insertion point should be placed at the end of the former selection. NSResponder declares, but doesn't implement this method.

moveForwardAndModifySelection:

– (void)**moveForwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection forward by one element or character. If the end being modified is the backward end, this method reduces the selection; if the end being modified is the forward end, it expands the selection. The first **moveBackwardAndModifySelection:** or **moveForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the forward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveLeft:

– (void)**moveLeft:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character to the left. In text, if there is a selection it should be deselected, and the insertion point should be placed at the left end of the former selection. NSResponder declares, but doesn't implement this method.

moveRight:

– (void)**moveRight:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character to the right. In text, if there is a selection it should be deselected, and the insertion point should be placed at the right end of the former selection. NSResponder declares, but doesn't implement this method.

moveToBeginningOfDocument:

– (void)**moveToBeginningOfDocument:**(id)*sender*

Implemented by subclasses to move the selection to the first element of the document, or the insertion point to the beginning. NSResponder declares, but doesn't implement this method.

moveToBeginningOfLine:

– (void)**moveToBeginningOfLine:**(id)*sender*

Implemented by subclasses to move the selection to the first element of the selected line, or the insertion point to the beginning of the line. NSResponder declares, but doesn't implement this method.

moveToBeginningOfParagraph:

– (void)**moveToBeginningOfParagraph:**(id)*sender*

Implemented by subclasses to move the insertion point to the beginning of the selected paragraph. NSResponder declares, but doesn't implement this method.

moveToEndOfDocument:

– (void)**moveToEndOfDocument:**(id)*sender*

Implemented by subclasses to move the selection to the last element of the document, or the insertion point to the end. NSResponder declares, but doesn't implement this method.

moveToEndOfLine:

– (void)**moveToEndOfLine:**(id)*sender*

Implemented by subclasses to move the selection to the last element of the selected line, or the insertion point to the end of the line. NSResponder declares, but doesn't implement this method.

moveToEndOfParagraph:

– (void)**moveToEndOfParagraph:**(id)*sender*

Implemented by subclasses to move the insertion point to the end of the selected paragraph. NSResponder declares, but doesn't implement this method.

moveUp:

– (void)**moveUp:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one element or character up. In text, if there is a selection it should be deselected, and the insertion point should be placed above the beginning of the former selection. NSResponder declares, but doesn't implement this method.

moveUpAndModifySelection:

– (void)**moveUpAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce the top or bottom end of the selection upward by one element, character, or line (whichever is appropriate for text direction). If the end being modified is the bottom, this method reduces the selection; if the end being modified is the top, it expands the selection. The first **moveDownAndModifySelection:** or **moveUpAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the top end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveWordBackward:

– (void)**moveWordBackward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one word backward. If there is a selection it should be deselected, and the insertion point should be placed at the end of the first word preceding the former selection. NSResponder declares, but doesn't implement this method.

moveWordBackwardAndModifySelection:

– (void)**moveWordBackwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection backward by one whole word. If the end being modified is the backward end, this method expands the selection; if the end being modified is the forward end, it reduces the selection. The first **moveWordBackwardAndModifySelection:** or **moveWordForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the backward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

moveWordForward:

– (void)**moveWordForward:**(id)*sender*

Implemented by subclasses to move the selection or insertion point one word forward. If there is a selection it should be deselected, and the insertion point should be placed at the beginning of the first word following the former selection. NSResponder declares, but doesn't implement this method.

moveWordForwardAndModifySelection:

– (void)**moveWordForwardAndModifySelection:**(id)*sender*

Implemented by subclasses to expand or reduce either end of the selection forward by one whole word. If the end being modified is the backward end, this method reduces the selection; if the end being modified is the forward end, it expands the selection. The first **moveWordBackwardAndModifySelection:** or **moveWordForwardAndModifySelection:** method in a series determines the end being modified by always expanding. Hence, this method results in the forward end becoming the mobile one if invoked first.

NSResponder declares, but doesn't implement this method.

nextResponder

– (NSResponder *)**nextResponder**

Returns the receiver's next responder, or **nil** if it has none.

See also: – **setNextResponder:**, – **noResponderFor:**

noResponderFor:

– (void)**noResponderFor:**(SEL)*eventSelector*

Handles the case where an event or action message falls off the end of the responder chain. NSResponder's implementation beeps if *eventSelector* is **keyDown:**.

pageDown:

– (void)**pageDown:**(id)*sender*

Implemented by subclasses to scroll the receiver down (or back) one page in its scroll view, also moving the insertion point to the top of the newly-displayed page. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageDown:**, – **scrollPageUp:**

pageUp:

– (void)**pageUp:**(id)*sender*

Implemented by subclasses to scroll the receiver up (or forward) one page in its scroll view, also moving the insertion point to the top of the newly-displayed page. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageDown:**, – **scrollPageUp:**

performKeyEquivalent:

– (BOOL)**performKeyEquivalent:**(NSEvent *)*theEvent*

Overridden by subclasses to handle a key equivalent. If the character code or codes in *theEvent* match the receiver's key equivalent, the receiver should respond to the event and return YES. NSResponder's implementation does nothing and returns NO.

Note: **performKeyEquivalent:** takes an NSEvent as its argument, while **performMnemonic:** takes an NSString containing the uninterpreted characters of the key event. You should extract the characters for a key equivalent using NSEvent's **charactersIgnoringModifiers** method.

See also: – **performKeyEquivalent:** (NSView, NSWindow, NSButton)

performMnemonic:

– (BOOL)**performMnemonic:**(NSString *)*aString*

Overridden by subclasses to handle a mnemonic. If the character code or codes in *theEvent* match the receiver's key equivalent, the receiver should respond to the event and return YES. NSResponder's implementation does nothing and returns NO.

See also: – **performMnemonic:** (NSView)

resignFirstResponder

– (BOOL)**resignFirstResponder**

Notifies the receiver that it's been asked to relinquish its status as first responder in its NSWindow. NSResponder's implementation returns YES, resigning first responder status. Subclasses can override this method to update state or perform some action such as unhighlighting the selection, or to return NO, refusing to relinquish first responder status.

Use NSWindow's **makeFirstResponder:**, not this method, to make an object the first responder. Never invoke this method directly.

See also: – **becomeFirstResponder**, – **acceptsFirstResponder**

rightMouseDown:

– (void)**rightMouseDown:**(NSEvent *)*theEvent*

Informs the receiver that the user has pressed the right mouse button. NSResponder's implementation simply passes this message to the next responder.

rightMouseDragged:

– (void)**rightMouseDragged:**(NSEvent *)*theEvent*

Informs the receiver that the user has moved the mouse with the right button pressed. NSResponder's implementation simply passes this message to the next responder.

rightMouseUp:

– (void)**rightMouseUp:**(NSEvent *)*theEvent*

Informs the receiver that the user has released the right mouse button. NSResponder's implementation simply passes this message to the next responder.

scrollLineDown:

– (void)**scrollLineDown:**(id)*sender*

Implemented by subclasses to scroll the receiver one line down in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageDown:**, – **lineScroll** (NSScrollView)

scrollLineUp:

– (void)**scrollLineUp:**(id)*sender*

Implemented by subclasses to scroll the receiver one line up in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **scrollPageUp:**, – **lineScroll** (NSScrollView)

scrollPageDown:

– (void)**scrollPageDown:**(id)*sender*

Implemented by subclasses to scroll the receiver one page down in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **pageDown:**, – **pageUp:**, – **pageScroll** (NSScrollView)

scrollPageUp:

– (void)**scrollPageUp:**(id)*sender*

Implemented by subclasses to scroll the receiver one page up in its scroll view, without changing the selection. NSResponder declares, but doesn't implement this method.

See also: – **pageDown:**, – **pageUp:**, – **pageScroll** (NSScrollView)

selectAll:

– (void)**selectAll:**(id)*sender*

Implemented by subclasses to select all selectable elements. NSResponder declares, but doesn't implement this method.

selectLine:

– (void)**selectLine:**(id)*sender*

Implemented by subclasses to select all elements in the line or lines containing the selection or insertion point. NSResponder declares, but doesn't implement this method.

selectParagraph:

– (void)**selectParagraph:**(id)*sender*

Implemented by subclasses to select all paragraphs containing the selection or insertion point. NSResponder declares, but doesn't implement this method.

selectToMark:

– (void)**selectToMark:**(id)*sender*

Implemented by subclasses to select all items from the insertion point or selection to a previously placed mark, including the selection itself if not empty. NSResponder declares, but doesn't implement this method.

See also: – **setMark:**, – **deleteToMark:**

selectWord:

– (void)**selectWord:**(id)*sender*

Implemented by subclasses to extend the selection to the nearest word boundaries outside it (up to, but not including, word delimiters). NSResponder declares, but doesn't implement this method.

setMark:

– (void)**setMark:**(id)*sender*

Implemented by subclasses to set a mark at the insertion point or selection, which is used by **deleteToMark:** and **selectToMark:**. NSResponder declares, but doesn't implement this method.

See also: – **swapWithMark:**

setMenu:

– (void)**setMenu:**(NSMenu *)*aMenu*

Sets the receiver's menu to *aMenu*. For NSApplication this is the same as the main menu, typically set using **setMainMenu:**.

See also: – **menu**

setNextResponder:

– (void)**setNextResponder:**(NSResponder *)*aResponder*

Sets the receiver's next responder to *aResponder*.

See also: – **nextResponder**

showContextHelp:

– (void)**showContextHelp:**(id)*sender*

Implemented by subclasses to invoke the host platform's help system, displaying information relevant to the receiver and its current state.

See also: – **helpRequested:**

swapWithMark:

– (void)**swapWithMark:**(id)*sender*

Swaps the mark and the selection or insertion point, so that what was marked is now the selection or insertion point, and what was the insertion point or selection is now the mark. NSResponder declares, but doesn't implement this method.

See also: – **setMark:**

transpose:

– (void)**transpose:**(id)*sender*

Transposes the characters to either side of the insertion point and advances the insertion point past both of them. Does nothing to a selected range of text. NSResponder declares, but doesn't implement this method.

transposeWords:

– (void)**transposeWords:**(id)*sender*

NSResponder declares, but doesn't implement this method.

tryToPerform:with:

– (BOOL)**tryToPerform:**(SEL)*anAction* **with:**(id)*anObject*

Attempts to perform the action method indicated by *anAction*. The method should take a single argument of type **id** and return **void**. If the receiver responds to *anAction*, it invokes the method with *anObject* as the argument and returns YES. If the receiver doesn't respond, it sends this message to its next responder with the same selector and object. Returns NO if no responder is found that responds to *anAction*.

See also: – **doCommandBySelector:**, – **sendAction:to:from:** (NSApplication)

uppercaseWord:

– (void)**uppercaseWord:**(id)*sender*

Implemented by subclasses to make uppercase every letter in the word or words surrounding the insertion point or selection, though without expanding the selection. If either end of the selection partially covers a word, that entire word is made uppercase. NSResponder declares, but doesn't implement this method.

See also: – **lowercaseWord:**, – **capitalizeWord:**, – **changeCaseOfLetter:**

validRequestorForSendType:returnType:

– (id)**validRequestorForSendType:**(NSString *)*sendType* **returnType:**(NSString *)*returnType*

Overridden by subclasses to determine what services are available. With each event, and for each service in the Services menu, the application object sends this message up the responder chain with the send and return type for the service being checked. This method is therefore invoked many times per event. If the receiver can place data of *sendType* on the pasteboard and receiver data of *returnType*, it should return **self**; otherwise it should return **nil**. NSResponder's implementation simply forwards this message to the next responder, ultimately returning **nil**.

Either *sendType* or *returnType*—but not both—may be empty. If *sendType* is empty, the service doesn't require input from the application requesting the service. If *returnType* is empty, the service, doesn't return data.

See “Services” in *OPENSTEP Programming Topics* for more information.

See also: – **registerServicesMenuSendType:returnTypes:** (NSApplication),
– **writeSelectionToPasteboard:types:** (NSApplication),
– **readSelectionFromPasteboard:** (NSApplication)

yank:

– (void)**yank:**(id)*sender*

Replaces the insertion point or selection with text from the kill buffer and selects that text. If invoked sequentially, cycles through the kill buffer in reverse order. See “Standard Action Methods for Selecting and Editing” in the class description for more information on the kill buffer. NSResponder declares, but doesn’t implement this method.

See also: – **deleteToBeginningOfLine:**, – **deleteToEndOfLine:**, – **deleteToBeginningOfParagraph:**,
– **deleteToEndOfParagraph:**, – **deleteToMark:**