NSControl

Inherits From:	NSView : NSResponder : NSObject
Conforms To:	NSCoding (NSResponder) NSObject (NSObject)
Declared In:	<AppKit/NSControl.h>

Class Description

NSControl is an abstract superclass that provides three fundamental features for implementing user-interface devices. First, as a subclass of NSView, NSControl draws, or coordinates the drawing of, the on-screen representation of the device. Second, it receives and responds to user-generated events within its bounds by overriding NSResponder's **mouseDown:** method and providing a position in the responder chain. Third, it implements the **sendAction:to:** method to send an action message to the NSControl's target object. Subclasses of NSControl defined in the Application Kit are NSBrowser, NSButton (and its subclass NSPopUpButton), NSColorWell, NSImageView, NSMatrix (and its subclass NSForm), NSScroller, NSSlider, NSTableView, and NSTextField. Instances of concrete NSControl subclasses are often referred to as, simply, *controls*.

Controls and Cells

Controls are usually associated with one or more cells—instances of a subclass of the abstract class NSCell. A control's cell (or cells) usually fit just inside the bounds of the control. Cells are objects that can draw themselves and respond to events, but they can do so only indirectly, upon instruction from their control, which acts as a kind of coordinating backdrop.

Controls manage the behavior of their cells. By inheritance from NSView, controls derive the ability for responding to user actions and rendering their on-screen representation. When users click on a control, it responds in part by sending **trackMouse:inRect:ofView:untilMouseUp:** to the cell that was clicked; upon receiving this message, the cell tracks the mouse and may have the control send the cell's action message to its target (either upon mouse-up or continuously, depending on the cell's attributes). When controls receive a display request, they, in turn, send their cell (or cells) a **drawWithFrame:inView:** message to have the cells draw themselves.

This relationship of control and cell makes two things possible: A control can manage cells of different types and with different targets and actions (see below); and a single control can manage muliple cells. Most Application Kit controls, like NSButtons and NSTextFields, manage only a single cell. But some controls, notably NSMatrix and NSForm, manage multiple cells (usually of the same size and attributes, and arranged in a regular pattern). Because cells are lighter-weight than controls, in terms of inherited data and behavior, it is more efficient to use a multi-cell control rather than multiple controls.

Many methods of NSControl—particularly methods that set or obtain values and attributes—have corresponding methods in NSCell. Sending a message to the control causes it to be forwarded to the control's cell *or* (if a multi-cell control) its selected cell. However, many NSControl methods are effective only in controls with single cells (these are noted in the method descriptions).

An NSControl subclass doesn't have to use an NSCell subclass to implement itself; NSScroller and NSColorWell are examples of NSControls that don't. However, such subclasses have to take care of details that NSCell would otherwise handle. Specifically, they have to override methods designed to work with a cell. What's more, the lack of a cell means you can't make use of NSMatrix capability for managing multi-cell arrays such as radio buttons.

Target and Action

Target objects and action methods (or messages) are part of the mechanism by which controls respond to user actions and enable users to communicate their intentions to an application. A target is an object that a control uses as the receiver of action messages. The target's class defines an action method to enable its instances to respond to these messages, which are sent as users click or otherwise manipulate the control. NSControl's **sendAction:to:** asks the NSApplication object, NSApp, to send an action message to the control's target object.

NSControl provides methods for setting and obtaining the target object and the action method. However, these methods require that an NSControl's cell (or cells) be cells that inherit from NSActionCell or custom cells that hold action and target as instance variables and can respond to the NSControl methods.

See the NSActionCell class specification for more on the implementation of target and action behavior, particularly how action messages with **nil** targets travel up the responder chain.

Field Validation and Entry Error-Handling

NSControl provides the delegation method **control:isValidObject:** for validating the contents of cells embedded in controls (instances of NSTextField and NSMatrix in particular). In validating you check for values that are permissible as objects, but that are undesirable in a given context, such as a date field in which dates should never be in the future, or zip codes that are valid for a certain state.

The method **control:isValidObject:** is invoked when the cursor leaves a cell (that is, the associated control relinquishes first-responder status) but before the string value of the cell's object is displayed. Return YES to allow display of the string and NO to reject display and return the cursor to the cell. The following example evaluates an object (an NSDate) and rejects it if the date is in the future:

```
- (BOOL)control:(NSControl *)control isValidObject:(id)obj
{
    if (control == contactsForm) {
        if (![obj isKindOfClass:[NSDate class]]) return NO;
        if ([[obj laterDate:[NSDate date]] isEqual:obj]) {
            NSRunAlertPanel(@"Date not valid",
                @"Reason: date in future", NULL, NULL, NULL);
```

```
            return NO;
        }
    }
    return YES;
}
```

NSControl also provides delegation methods that are invoked when formatters for a control's cells cannot format a string **(control:didFailToFormatString:errorDescription:**) or reject a partial string entry (**control:didFailToValidatePartialString:errorDescription:**). It also provides **control:textView:doCommandBySelector:**, which allows delegates the opportunity to detect and respond to key bindings, such as **complete:** (name completion).

Changing the NSCell Class

Since NSControl uses objects derived from the NSCell class to implement most of its actual functionality, you can usually implement a unique user interface device by creating a subclass of NSCell rather than NSControl. As an example, let's say you want all your application's NSSliders to have a type of cell other than the generic NSSliderCell. First, you create a subclass of NSCell, NSActionCell, or NSSliderCell. (Let's call it MyCellSubclass.) Then, you can simply invoke NSSlider's **setCellClass:** class method:

```
[NSSlider setCellClass:[MyCellSubclass class]];
```

All NSSliders created thereafter will use MyCellSubclass, until you call **setCellClass:** again.

If you want to create generic NSSliders (ones that use NSSliderCell) in the same application as the customized NSSliders that use MyCellSubclass, there are two possible approaches. One is to invoke **setCellClass:** as above whenever you're about to create a custom NSSlider, resetting the cell class to NSSliderCell afterwards. The other approach is to create a custom subclass of NSSlider that automatically uses MyCellSubclass, as explained below.

Creating New NSControls

If you create a custom NSControl subclass that uses a custom subclass of NSCell, you should override NSControl**'s cellClass** method:

```
+ (Class) cellClass
{
    return [MyCellSubclass class];
}
```

NSControl's **initWithFrame:** method will use the return value of **cellClass** to allocate and initialize an NSCell of the correct type.

Override the designated initializer (**initWithFrame:**) if you create a subclass of NSControl that performs its own initialization.

Method Types

Initializing an NSControl	– initWithFrame:
Setting the control's cell	+ cellClass + setCellClass: – cell – setCell:
Enabling and disabling the control	– isEnabled – setEnabled:
Identifying the selected cell	– selectedCell – selectedTag
Setting the control's value	– doubleValue – setDoubleValue: – floatValue – setFloatValue: – intValue – setIntValue: – objectValue – setObjectValue: – stringValue – setStringValue: – setNeedsDisplay
Interacting with other controls	– takeDoubleValueFrom: – takeFloatValueFrom: – takeIntValueFrom: – takeObjectValueFrom: – takeStringValueFrom:
Formating text	– alignment – setAlignment: – font – setFont: – setFloatingPointFormat:left:right:
Managing the field editor	– abortEditing – currentEditor – validateEditing
Resizing the control	– calcSize – sizeToFit

Displaying a cell	– selectCell: – drawCell: – drawCellInside: – updateCell: – updateCellInside:
Implementing the target/action mechanism	– action – setAction: – target – setTarget: – isContinuous – setContinuous: – sendAction:to: – sendActionOn:
Getting and setting attributed-string values	– attributedStringValue – setAttributedStringValue:
Getting and setting tags	– tag – setTag:
Simulating mouse clicks	– performClick:sender:
Tracking the mouse	– mouseDown: – ignoresMultiClick – setIgnoresMultiClick:

Class Methods

cellClass

+ (Class)**cellClass**

Returns the class of cells used by the receiving class (which must be NSControl or one of its subclasses). Returns **nil** if no cell class has been specified for the receiving class or any of its superclasses (up to NSControl).

See also: – **cell**, – **setCell:**

setCellClass:

+ (void)**setCellClass:**(Class)*class*

Sets the class of cells used by instances of the receiver, which must be the NSControl class or one of its subclasses.

See also: – **cell**, – **setCell:**

Instance Methods

abortEditing

– (BOOL)**abortEditing**

Terminates and discards any editing of text displayed by the receiving control and removes the field editor's delegate. Returns YES if there was a field editor associated with the control, NO otherwise.

See also: – **currentEditor**, – **validateEditing**

action

– (SEL)**action**

Returns the action-message selector of the receiver's cell (the default NSControl behavior), or the default action-message selector for a control with multiple cells (such as an NSMatrix or an NSForm). For controls with multiple cells, it's better to get the action-message selector for a particular cell, for instance:

```
someAction = [[theControl selectedCell] action];
```

See also: – **setAction:**, – **setTarget:**, – **target**

alignment

– (NSTextAlignment)**alignment**

Returns the alignment mode of the text in the receiver's cell. The return value can be one of these constants: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, or NSNaturalTextAlignment (the default alignment).

See also: – **setAlignment:**

attributedStringValue

– (NSAttributedString *)**attributedStringValue**

Returns the object value of the receiver's cell (or selected cell) as an attributed string after validating any editing currently being done. If no cell is associated with the receiver, returns an empty attributed string.

See also: – **setAttributedStringValue:**

calcSize

– (void)**calcSize**

Recomputes any internal sizing information for the NSControl, if necessary, by invoking its NSCell's **calcDrawInfo:** method. Most NSControls maintain a flag that informs them if any of their cells have been modified in such a way that the location or size of the cell should be recomputed. If this happens, **calcSize** is automatically invoked whenever the NSControl is displayed; you never need to invoke it yourself.

See also: – **calcSize** (NSMatrix, NSForm), – **sizeToFit**

cell

– (id)**cell**

Returns the receiver's cell. In NSControls with multiple cells (such as NSMatrix or NSForm), use **selectedCell** or a similar method for finding a particular cell.

See also: + **cellClass**, – **setCell:**, + **setCellClass:**

currentEditor

– (NSText *)**currentEditor**

If the receiving NSControl is being edited—that is, it has an NSText object acting as its field editor, and is the first responder of its NSWindow—this method returns the NSText editor; otherwise, it returns **nil**.

See also: – **abortEditing**, – **validateEditing**

doubleValue

– (double)**doubleValue**

Returns the value of the receiver's cell as a double-precision floating point number. If the NSControl contains many cells (for example, NSMatrix), then the value of the currently **selectedCell** is returned. If

the NSControl is in the process of editing the affected Cell, then **validateEditing** is invoked before the value is extracted and returned.

See also: – **floatValue**, – **intValue**, – **objectValue**, – **setDoubleValue:**, – **stringValue**

drawCell:

– (void)**drawCell:**(NSCell *)*aCell*

If *aCell* is the cell used to implement this NSControl, then the NSControl is displayed. This method is provided primarily to support a consistent set of methods between NSControls with single and multiple cells, since a NSControl with multiple cells needs to be able to draw a single cell at a time.

See also: – **selectCell:**, – **updateCell:**, – **updateCellInside:**

drawCellInside:

– (void)**drawCellInside:**(NSCell *)*aCell*

Draws the inside of the receiver's cell (the area within a bezel or border). If the NSControl is transparent, the method causes the superview to draw itself. This method invokes NSCell's **drawInteriorWithFrame:inView:** method. This method has no effect on NSControls (such as NSMatrix and NSForm) that have multiple cells.

See also: – **selectCell:**, – **updateCell:**, – **updateCellInside:**

floatValue

– (float)**floatValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as a single-precision floating point number. See **doubleValue** for more details.

See also: – **doubleValue**, – **intValue**, – **objectValue**, – **setFloatValue:**, – **stringValue**

font

– (NSFont *)**font**

Returns the NSFont used to draw text in the receiver's cell.

See also: – **setFont:**

ignoresMultiClick

– (BOOL)**ignoresMultiClick**

Returns whether the receiving NSControl ignores multiple clicks made in rapid succession. See **setIgnoresMultiClick:** for details.

initWithFrame:

– (id)**initWithFrame:**(NSRect)*frameRect*

Initializes and returns a new NSControl object in *frameRect*, and creates a cell for it if the cell's class has been specified for controls of this type with **setCellClass:**. Because NSControl is an abstract class, invocations of this method should appear only in the designated initializers of sublcasses; that is, there should always be a more specific designated initializer for the subclass, as this **initWithFrame:** is the designated initializer for NSControl.

intValue

– (int)**intValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as an integer. See **doubleValue** for more details.

See also: – **floatValue**, – **doubleValue**, – **objectValue**, – **setIntValue:**, – **stringValue**

isContinuous

– (BOOL)**isContinuous**

Returns whether the control's NSCell continuously sends its action message to its target during mouse tracking.

See also: – **setContinous:**

isEnabled

– (BOOL)**isEnabled**

Returns whether the receiver reacts to mouse events.

See also: – **setEnabled:**

mouseDown:

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

Invoked when the mouse button is pressed while the cursor is within the bounds of the NSControl. This method highlights the NSControl's NSCell and sends it a **trackMouse:inRect:ofView:untilMouseUp:** message. Whenever the NSCell finishes tracking the mouse (for example, because the cursor has left the cell's bounds), the cell is unhighlighted. If the mouse button is still down and the cursor reenters the bounds, the cell is again highlighted and a new **trackMouse:inRect:ofView:untilMouseUp:** message is sent. This behavior repeats until the mouse button goes up. If it goes up with the cursor in the control, the state of the control is changed, and the action message is sent to the target. If the mouse button goes up when the cursor is outside the control, no action message is sent.

See also: – **ignoresMultiClick**, – **trackMouse:inRect:ofView:untilMouseUp:**(NSCell)

objectValue

– (id)**objectValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as an Objective-C object. See **doubleValue** for more details.

See also: – **floatValue**, – **doubleValue**, – **intValue**, – **setObjectValue:**, – **stringValue**

performClick:

– (void)**performClick:***sender*

Programmatically simulates a mouse click on the receiver's cell, including the invocation of the action method in the target object. Raises an exception if the action message cannot be successfully sent.

selectCell:

– (void)**selectCell:**(NSCell *)*aCell*

If *aCell* is a cell of the receiving NSControl and is unselected, this method selects *aCell* (by setting its state to YES) and redraws the NSControl.

See also: – **selectedCell**

selectedCell

– (id)**selectedCell**

Returns the receiver's selected cell. The default implementation for NSControl simply returns the associated cell (or **nil** if no cell has been set). Subclasses of NSControl that manage multiple cells (such as NSMatrix and NSForm) override this method to return the cell selected by users.

See also: – **cell**, – **setCell:**

selectedTag

– (int)**selectedTag**

Returns the tag integer of the receiver's selected cell (see **selectedCell**) or -1 if there is no selected cell. When you set the tag of an control with a single cell in Interface Builder, it sets the tags of both the control and the cell with the same value as a convenience.

See also: – **setTag:**, – **tag**

sendAction:to:

– (BOOL)**sendAction:**(SEL)*theAction* **to:**(id)*theTarget*

Sends **sendAction:to:from:** to NXApp, which in turn sends a message to *theTarget* to perform *theAction*, adding the receiver as the argument to the **from:** keyword. **sendAction:to:** is invoked primarily by NSCell's **trackMouse:inRect:ofView:untilMouseUp:**.

If *theAction* is **nil**, no message is sent. If *theTarget* is **nil**, NXApp looks for an object that can respond to the message by following the responder chain (see the class description for NSActionCell). This method returns **nil** if no object that responds to *theAction* could be found.

See also: – **action**, – **target**

sendActionOn:

– (int)**sendActionOn:**(int)*mask*

Sets the conditions on which the receiver sends action messages to its target (continuously, mouse up, and others) and returns a bit mask with which to detect the previous settings. NSControl's default implementation simply invokes the **sendActionOn:** method of its associated cell

See also: – **sendAction:to:**, – **sendActionOn:**(NSCell)

setAction:

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

Sets the NSControl's action method to *aSelector*. If *aSelector* is **nil**, then no action messages will be sent from the NSControl.

See also: – **action**, – **setTarget:**, – **target**

setAlignment:

– (void)**setAlignment:**(NSTextAlignment)*mode*

Sets the alignment of text in the receiver's cell and, if the cell is being edited, aborts editing and updates the cell. *mode* is one of five constants: NSLeftTextAlignment, NSRightTextAlignment, NSCenterTextAlignment, NSJustifiedTextAlignment, NSNaturalTextAlignment (the default alignment for the text).

See also: – **alignment**

setAttributedStringValue:

– (void)**setAttributedStringValue:**(NSAttributedString *)*object*

Sets the value of the receiver's cell (or selected cell) as an attributed string. If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks it for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – attributedStringValue

setCell:

– (void)**setCell:**(NSCell *)*aCell*

Sets the receiver's cell to *aCell*. Use this method with great care as it can irrevocably damage the affected control; specifically, you should only use this method in initializers for subclasses of NSControl.

See also: – **cell**, – **selectedCell**

setContinuous:

– (void)**setContinuous:**(BOOL)*flag*

Sets whether the receiver's cell continuously sends its action message to its target as it tracks the mouse.

See also: – **isContinuous**

setDoubleValue:

– (void)**setDoubleValue:**(double)*aDouble*

Sets the value of the receiver's cell (or selected cell) as aDouble (a double-precision floating point number). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **doubleValue**,– **setFloatValue:**, – **setIntValue:**, – **setObjectValue:**, – **setStringValue:**

setEnabled:

– (void)**setEnabled:**(BOOL)*flag*

Sets whether the receiving NSControl's cell—or if there is no associated cell, the NSControl itself—is active (that is, whether it tracks the mouse and sends its action to its target). If flag is NO, any editing is aborted. Redraws the entire Control if autodisplay is enabled. Subclasses may want to override this method to redraw only a portion of the control when the enabled state changes, as do NSButton and NSSlider.

See also: – **isEnabled**

setFloatValue:

– (void)**setFloatValue:**(float)*aFloat*

Sets the value of the receiver's cell (or selected cell) as *aFloat* (a single-precision floating point number). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **floatValue**, – **setDoubleValue:**, – **setIntValue:**, – **setObjectValue:**, – **setStringValue:**

setFloatingPointFormat:left:right:

– (void)**setFloatingPointFormat:**(BOOL)*autoRange*
left:(unsigned)*leftDigits*
right:(unsigned)*rightDigits*

Sets the autoranging and floating point number format of the receiver's cell, so that at most *leftDigits* are displayed to the left of the decimal point, and *rightDigits* to the right. See the description of this method in the NSCell class specification for details. If the cell is being edited, what's typed is discarded and the cell's interior is redrawn.

See also: – **setFloatingPointFormat:left:right:**(NSCell)

setFont:

– (void)**setFont:**(NSFont *)*fontObject*

Sets the font used to draw text in the receiver's cell to *fontObject*. If the cell is being edited, the text in the cell is redrawn in the new font and the cell's editor (the NSText object used globally for editing) is updated with the new NSFont.

See also: – setFont:

setIgnoresMultiClick:

– (void)**setIgnoresMultiClick:**(BOOL)*flag*

Sets whether the receiving NSControl ignores multiple clicks made in rapid succession. By default, controls treat double-clicks as two distinct clicks, triple-clicks as three distinct clicks, and so on. However, when an NSControl returning YES to this method receives multiple clicks (within a predetermined interval), each **mouseDown** event after the first is passed on to **super**.

See also: – **ignoresMultiClick**

setIntValue:

– (void)**setIntValue:**(int)*anInt*

Sets the value of the receiver's cell (or selected cell) as an integer (*anInt*). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **intValue**,– **setDoubleValue:**, – **setFloatValue:**, – **setObjectValue:**, – **setStringValue:**

setNeedsDisplay

– (void)**setNeedsDisplay**

Marks the receiving NSControl as needing redisplay (assuming automatic display is enabled) after recalculation of its dimensions.

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

setObjectValue:

– (void)**setObjectValue:**(id)*object*

Sets the value of the receiver's cell (or selected cell) as an Objective-C object. If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **objectValue**, – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**, – **setStringValue:**

setStringValue:

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

Sets the value of the receiver's cell (or selected cell) as an NSString object (*aString*). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See also: – **setDoubleValue:**, – **setFloatValue:**, – **setIntValue:**, – **setObjectValue:**, – **stringValue**

setTag:

– (void)**setTag:**(int)*anInt*

Sets the tag of the receiving NSControl to *anInt*. It doesn't affect the tag of the receiver's cell.

See also: – **tag**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the target object for the action message of the receiver's cell; NSCell's **setTarget:** is used instead of any subclass override of this method. If *anObject* is **nil** and the control sends an action message, the application looks for an object that can respond to the message by following the responder chain (see description of the NSActionCell class for details).

See also: – **action**, – **setAction:**, – **target**, – **setTarget:**(NSCell)

sizeToFit

– (void)**sizeToFit**

Changes the width and the height of the receiver's frame so that they are the minimum needed to contain its cell. If you want a multiple-cell custom subclass of NSControl to size itself to fit its cells, you must override this method.

See also: – **calcSize**

stringValue

– (NSString *)**stringValue**

Returns the value of the receiver's cell (or selected cell, if a multiple-cell NSControl) as an NSString object. See **doubleValue** for details.

See also: – **floatValue**, – **doubleValue**, – **intValue**, – **objectValue**, – **setStringValue:**

tag

– (int)**tag**

Returns the tag identifying the receiving control (not the tag of the receiver's cell).

See also: – **setTag:**

takeDoubleValueFrom:

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

Sets the double-precision floating-point value of the receiving control's cell (or selected cell) to the value obtained by sending a **doubleValue** message to *sender*. You can use this method to link action messages between controls. It permits one control or cell (*sender*) to affect the value of another control (the receiver) by invoking this method in an action message to the receiver. For example, a text field can be made the target of a slider. Whenever the slider is moved, it will send a **takeDoubleValueFrom:** message to the text field. The text field will then get the slider's floating-point value, turn it into a text string, and display it, thus tracking the value of the slider.

takeFloatValueFrom:

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

Sets the receiving NSControl's selected cell to the value obtained by sending a **floatValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

takeIntValueFrom:

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

Sets the receiving NSControl's selected cell to the value obtained by sending a **intValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

takeObjectValueFrom:

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

Sets the receiving NSControl's selected cell to the value obtained by sending a **objectValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

takeStringValueFrom:

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

Sets the receiving NSControl's selected cell to the value obtained by sending a **stringValue** message to another control or cell (*sender*). See **takeDoubleValueFrom:** for more information.

target

– (id)**target**

Returns the target object of the receiver’s cell.

See also: – **action**, – **setAction:**, – **setTarget:**

updateCell:

– (void)**updateCell:**(NSCell *)*aCell*

Redisplays *aCell* or marks it for redisplay.

updateCellInside:

– (void)**updateCellInside:**(NSCell *)*aCell*

Redisplays the inside of *aCell* or marks it for redisplay.

validateEditing

– (void)**validateEditing**

Validates the user's changes to text in a cell of the receiving control. Validation sets the object value of the cell to the current contents of the cell's editor (the NSText object used for editing), storing its a simple NSString or an attributed string object based on the attributes of the editor.

See also: – **abortEditing**, – **currentEditor**

Methods Implemented By the Delegate

control:didFailToFormatString:errorDescription:

– (BOOL)**control:**(NSControl *)*control*
didFailToFormatString:(NSString *)*string*
errorDescription:(NSString *)*error*

Invoked when the formatter for *control*'s cell (or selected cell) cannot convert an NSString (*string)* to an underlying object. *error* is a localized user-presentable NSString that explains why the conversion failed. Evaluate the error or query the user and return YES if *string* should be accepted as-is, or NO if *string* should be rejected.

See also: – **getObjectValue:forString:errorDescription:**(NSFormatter)

control:didFailToValidatePartialString:errorDescription:

– (void)**control:**(NSControl *)*control*
didFailToValidatePartialString:(NSString *)*string*
errorDescription:(NSString *)*error*

Invoked when the formatter for *control*'s cell (or selected cell) rejects a partial string a user is typing into the cell. This NSString (*string*) includes the character that caused the rejection. *error* is a localized user-presentable NSString that explains why the validation failed. Evaluate the error or query the user and return YES if *string* should be accepted as-is, or NO if *string* should be rejected.

See also: – **isPartialStringValid:newEditingString:errorDescription:**(NSFormatter)

control:isValidObject:

– (BOOL)**control:**(NSControl *)*control* **isValidObject:**(id)*object*

Invoked when the cursor leaves a cell but before the string value of the cell's object is displayed. Return YES to allow display of the string and NO to reject display and return the cursor to the cell. This method gives the delegate the opportunity to validate the contents of *control*'s cell (or selected cell). In validating,

the delegate checks *object* to determine if it falls within a permissible range, has required attributes, accords with a given context, and so on. An example of an object subject to such and evaluation is an NSDate object which should not represent a future date, or a monetary amount (represented by an NSNumber) that exceeds a predetermined limit.

control:textShouldBeginEditing:

– (BOOL)**control:**(NSControl *)*control* **textShouldBeginEditing:**(NSText *)*fieldEditor*

Sent directly by *control* to the delegate when the cursor tries to enter a cell of the control that allows editing of text (such as a text field or form field). Return YES if the NSControl's *fieldEditor* should be allowed to start editing the text, NO otherwise.

control:textShouldEndEditing:

– (BOOL)**control:**(NSControl *)*control* **textShouldEndEditing:**(NSText *)*fieldEditor*

Sent directly by *control* to the delegate when the cursor tries to leave a a cell of the control that allows editing of text (such as a text field or a form field). Return YES if the control's *fieldEditor* should be allowed to end its edit session, NO otherwise.

control:textView:doCommandBySelector:

– (BOOL)**control:**(NSControl *)*control*
textView:(NSTextView *)*textView*
doCommandBySelector:(SEL)*command*

Invoked when users press keys with predefined bindings in *control*'s cell or selected cell, as communicated to the control by the cell's field editor (*textView*). The delegate returns YES if it handles the key binding, and NO otherwise. These bindings are usually implemented as methods (*command*) defined in NSResponder; examples of such key bindings are arrow keys (for directional movement) and the Escape key (for name completion). By implementing this method, the delegate can override the default implementation of *command* and supply its own behavior.

For example, the default method for completing partially typed pathnames or symbols (usually when users press the Escape key) is **complete:**. The default implementation of **complete:** (in NSResponder) does nothing. The delegate could evaluate *command* and, if it's **complete:**, get the current string from *textView* and then expand it, or display a list of potential completions, or do whatever else is appropriate.

controlTextDidBeginEditing:

– (void)**controlTextDidBeginEditing:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate and all observers of the notification when a control with editable cells (such as a text field, form field, or an NSMatrix) begins editing text. The name of the notification (*aNotification*) is always NSControlTextDidBeginEditingNotification. Use the key @"NSFieldEditor" to obtain the field editor from *aNotification*'s **userInfo** dictionary. If the delegate implements this method, it's automatically registered to receive this notification.

controlTextDidEndEditing:

– (void)**controlTextDidEndEditing:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate and all observers of the notification when a control with editable cells (such as a text field, form field, or an NSMatrix) ends editing text. The name of the notification (*aNotification*) is always NSControlTextDidEndEditingNotification. Use the key @"NSFieldEditor" to obtain the field editor from *aNotification*'s **userInfo** dictionary. If the delegate implements this method, it's automatically registered to receive this notification.

controlTextDidChange:

– (void)**controlTextDidChange:**(NSNotification *)*aNotification*

Sent by the default notification center to the delegate when the text in the receiving control (usually a text field, form, or NSMatrix with editable cells) changes. The name of the notification *aNotification* is always NSControlTextDidChangeNotification. Use the key @"NSFieldEditor" to obtain the field editor from *aNotification*'s **userInfo** dictionary. If the delegate implements this method, it's automatically registered to receive this notification.

Notifications

NSControl posts the following notifications to interested observers and its delegate.

NSControlTextDidBeginEditingNotification

Notification Object The NSControl posting the notification. The field editor of the edited cell originally sends a NSTextDidBeginEditingNotification to the control, which passes it on in this form to its delegate.

User Info

Key	Value
@"NSFieldEditor"	The edited cell's field editor

See description of **controlTextDidBeginEditing:**, above, for details.

NSControlTextDidChangeNotification

Notification Object The NSControl posting the notification. The field editor of the edited cell originally sends a NSTextDidChangeNotification to the control, which passes it on in this form to its delegate.

User Info

Key	Value
@"NSFieldEditor"	The edited cell's field editor

See description of **controlTextDidChange:**, above, for details.

NSControlTextDidEndEditingNotification

Notification Object The NSControl posting the notification. The field editor of the edited cell originally sends a NSTextDidEndEditingNotification to the control, which passes it on in this form to its delegate.

User Info

Key	Value
@"NSFieldEditor"	The edited cell's field editor

See description of **controlTextDidEndEditing:**, above.