# NSWindow

**Inherits From:** NSResponder : NSObject

**Conforms To:** NSCoding (NSResponder)
NSObject (NSObject)

**Declared In:** AppKit/NSWindow.h

## Class at a Glance

### Purpose

An NSWindow manages an on-screen window, coordinating the display and event handling for its NSViews. Interface Builder allows you to create and set up NSWindows, but there are many things you may wish to do programmatically as well.

### Principal Attributes

- Manages a view hierarchy
- Distributes events to view objects
- Uses a delegate
- Provides a field editor to view objects

### Creation

Interface Builder

– initWithContentRect:styleMask:backing:defer: Designated initializer.

### Commonly Used Methods

| | |
|---|---|
| – makeKeyAndOrderFront: | Move the NSWindow to the front and makes it the key window. |
| – makeFirstResponder: | Sets the first responder in the NSWindow. |
| – fieldEditor:forObject: | Returns the shared text object for the NSWindow. |
| – setContentView: | Sets the root-level NSView in the NSWindow. |
| – representedFilename | Returns the filename whose contents the NSWindow presents. |
| – setDocumentEdited: | Sets whether the NSWindow's represented file needs to be saved. |
| – setTitle: | Sets the title of the NSWindow. |
| – setTitleWithRepresentedFilename: | Sets the title of the NSWindow in a readable format for filenames. |

## Class Description

The NSWindow class defines objects that manage and coordinate the windows that an application displays on the screen. A single NSWindow object corresponds to at most one on-screen window. The two principal functions of NSWindow are to provide an area in which NSViews can be placed, and to accept and distribute, to the appropriate NSViews, events that the user instigates through actions on the mouse and keyboard. Note that the term *window* sometimes refers to the Application Kit object and sometimes to the Window Server's PostScript window device; which meaning is intended is made clear in context. The Application Kit also defines an abstract subclass of NSWindow—NSPanel—that adds behavior more appropriate for auxiliary windows.

You typically set windows up using Interface Builder, which allows you to position them, set up many of their visual and behavioral attributes, and lay out views on them. The programmatic work you do with windows more often involves bringing them on and off the screen; changing dynamic attributes such as the window's title; running modal windows to restrict user input; and assigning a delegate that can monitor certain of its actions, such as closing and resizing.

### Window Anatomy

An NSWindow is defined by a *frame rectangle* that encloses the entire window, including its title bar, border, and other peripheral elements (such as the resize bar on OPENSTEP for Mach), and by a *content rectangle* that encloses just its content area. Both rectangles are specified in the screen coordinate system and restricted to integer values. The frame rectangle establishes the NSWindow's *base coordinate system*. This coordinate system is always aligned with and measured in the same increments as the screen coordinate system (in other words, the base coordinate system can't be rotated or scaled). The origin of the base coordinate system is the bottom left corner of the NSWindow's frame rectangle.

You create an NSWindow programmatically through one of the **initWithContentRect:...** methods by specifying, among other attributes, the size and location of its content rectangle. The frame rectangle is derived from the dimensions of the content rectangle. Various sections below describe other attributes you can specify at initialization and afterward.

When it's created, an NSWindow automatically creates two NSViews: An opaque *frame view* that fills the frame rectangle and draws the border, title bar, other peripheral elements, and background, and a transparent *content view* that fills the content rectangle. The frame view and its peripheral elements are private objects that your application can't access directly. The content view is the "highest" accessible NSView in the NSWindow; you can replace the default content view with an NSView of your own creation using the **setContentView:** method. The NSWindow determines the placement of the content view; you can't position it using NSView's **setFrame...** methods, but must use NSWindow's placement methods, described below under "Windows on the Screen."

You add other NSViews to the NSWindow as subviews of the content view, or as subviews of any of the content view's subviews, and so on, through NSView's **addSubview:** method. This tree of NSViews is called the NSWindow's *view hierarchy*. When an NSWindow is told to display itself, it does so by sending **display...** messages to the top-level NSView in its view hierarchy. Because displaying is carried out in a

determined order, the content view (which is drawn first) may be wholly or partially obscured by its subviews, and these subviews may be obscured by their subviews (and so on).

#### Window Styles

The peripheral elements that an NSWindow displays define its style. Though you can't access and manipulate them directly, you can determine at initialization whether an NSWindow has them by providing a style mask to the **initWithContentRect:styleMask:backing:defer:** method. There are four possible style elements, specifiable by combining their mask values using the C bitwise OR operator:

| Element | Mask Value |
|---|---|
| A title bar | NSTitledWindowMask |
| A close button | NSClosableWindowMask |
| A miniaturize button | NSMiniaturizableWindowMask |
| A resize bar, border, or box | NSResizableWindowMask |

You can also specify NSBorderlessWindowMask, in which case none of these accoutrements is used.

## Windows on the Screen

NSWindows can be placed on the screen in three dimensions. Besides horizontal and vertical placement, NSWindows are ordered back-to-front in several distinct *levels*, which group windows of similar type and purpose so that the more "important" ones appear above those less so. Placing an NSWindow on the screen is accomplished with the **setFrame:display:** method and its variants, **setFrameOrigin:** and **setFrameTopLeftPoint:**. Ordering takes place in two ways. **setLevel:** puts an NSWindow into a group, such as that for standard windows, floating windows (for example, palettes and some inspector panels), menus, and so on. **orderWindow:relativeTo:** orders an NSWindow within its level above or below another. Convenience methods for ordering include **makeKeyAndOrderFront:**, **orderFront:** and **orderBack:**, as well as **orderOut:**, which removes an NSWindow from the screen. The **isVisible** method tells whether an NSWindow is on-screen or off. You can also set a window to order out automatically when its application isn't active using **setHidesOnDeactivate:**.

NSWindow offers several means of constraining and adjusting window placement. **setMinSize:** and **setMaxSize:** limit the user's ability to resize the NSWindow (you can still set it to any size programmatically). Similarly, **setAspectRatio:** keeps a window's width and height at the same proportions as the user resizes it, and **setResizeIncrements:** makes the window resize in discrete amounts larger than a single pixel. **constrainFrameRectToScreen:** adjusts a proposed frame rectangle so that it lies on the screen in such a way that the user can move and resize a window. Note that any NSWindow with a title bar automatically constrains itself to the screen. **cascadeTopLeftFromPoint:** shifts the top left point by an amount that allows one NSWindow to be placed relative to another so that both their title bars are visible. Finally, the **center** method places an NSWindow in the most prominent location on the screen, one suitable for important messages and attention panels.

Closely related to window ordering is the idea of opening or closing an NSWindow. Normally, opening is accomplished simply by ordering the NSWindow above or below another that's on-screen. Closing a window involves explicit use of the either the **close** method, which simply removes the NSWindow from the screen, or **performClose:**, which highlights the close button as though the user clicked it. Closing an NSWindow involves at least ordering it out, but adds the possibility of disposing of it altogether. The **setReleasedWhenClosed:** method sets whether an NSWindow releases itself when sent a close message. An NSWindow's delegate is also notified when it's about to close, as described under "Notifications and the NSWindow's Delegate."

Miniaturizable windows can be removed from the screen and replaced by a smaller counterpart, whether a freestanding miniwindow or, on Microsoft Windows, a button in the task bar. The **miniaturize:** and **deminiaturize:** methods reduce and reconstitute an NSWindow, and **performMiniaturize:** simulates the user clicking on the NSWindow's miniaturize button. You can also set the image and title displayed in a freestanding miniwindow by sending **setMiniwindowImage:** and **setMiniwindowTitle:** messages to the NSWindow object.

An NSWindow can store its placement in the user defaults system, so that it appears in the same location the next time the user starts the application. The **saveFrameUsingName:** method stores the frame rectangle, and **setFrameUsingName:** sets it from the value in user defaults. You can also use the **setFrameAutosaveName:** method to have an NSWindow save the frame rectangle any time it changes. To expunge a frame rectangle from the defaults system, use the class method **removeFrameUsingName:**.

## Titles and Represented Files

A titled NSWindow can display an arbitrary title or one derived from a filename. **setTitle:** puts an arbitrary string on the title bar. **setTitleWithRepresentedFilename:** formats a filename in the title bar in a readable format (which varies with the platform), and associates the NSWindow with that file. You can set the associated file without changing the title using **setRepresentedFilename:**. You can use the association between the NSWindow and the file in any way you see fit. One convenience offered by NSWindow is marking the file as being edited, so that you can prompt the user to save it on closing the window. The method for marking this is **setDocumentEdited:**. When the window closes, its delegate can check it with **isDocumentEdited** to see whether the document needs to be saved.

Most OPENSTEP applications include a submenu that displays the titles of windows, called the *Window menu*. This submenu automatically lists windows that have a title bar and are resizable, and that can become the main window (as described under "Event Handling"). When you change an NSWindow's title, this change is also automatically reflected in the Window menu. You can exclude a window that would otherwise be listed by sending it a **setExcludedFromWindowsMenu:** message.

## Window Device Attributes

Nearly every NSWindow has a corresponding PostScript window device in the Window Server. The window device holds the NSWindow's drawn image, and has two attributes determined by the Window Server and five attributes that the NSWindow controls. The Window Server assigns the window device a

unique identifier (within an application). This is the *window number*, and it's returned by the **windowNumber** method. Each window also has a PostScript graphics state that most NSViews share for drawing (NSViews can create their own as well). The **gstate** method returns its identifier. The five attributes under direct NSWindow control are:

- Where the drawn image is stored, called the window's *backing*
- When the window device is created
- Whether the window device persists when the window is off-screen
- How much memory is used for each pixel (also called the *depth limit*)
- Whether the depth limit changes with the screen capacity

A window device's backing is set when the NSWindow is initialized, and can be one of three types. A *buffered* window device renders all drawing into a display buffer and then flushes it to the screen. This produces very smooth display, but can require significant amounts of memory. Buffered windows are best for displaying material that must be redrawn often, such as text. A *retained* window device also uses a buffer, but draws directly to the screen where possible and to the buffer for any portions that are obscured. A *nonretained* window device has no buffer at all, and must redraw portions as they're exposed. Further, this redrawing is suspended when the NSWindow's display mechanism is preempted. For example, if the user drags a window across a nonretained window, the nonretained window is "erased" and isn't redrawn until the user releases the mouse. Both retained and nonretained windows are also subject to a flashing effect as individual drawing operations are performed, but their results do get to the screen more quickly than those of buffered windows. You can change the backing type between buffered and retained after initialization using the **setBacking:** method.

The last argument to **initWithContentRect:styleMask:backing:defer:** specifies whether the NSWindow creates its window device immediately, or only when it's ordered on-screen. Deferring creation of the window device can offer some performance gain for windows that aren't displayed immediately, as it reduces the amount of work that needs to be performed up front. This is particularly useful when creation of the NSWindow itself can't be deferred or when an NSWindow is needed for purposes other than displaying content. Submenus with key equivalents, for example, must exist for the key equivalents to work, but may never actually be displayed.

Memory can also be saved by destroying the window device when the NSWindow is ordered off-screen. The **setOneShot:** method controls this behavior. One-shot window devices exist only when their NSWindows are ordered on-screen.

Like the display hardware, a window device's buffer has a depth, or a limit to the memory allotted each pixel. Buffered and retained windows start out with a default window depth of 2 bits per pixel, and this depth grows to the window device's limit as the NSWindow draws richer images (more shades of gray, more colors). A window device's depth is set using the **setDepthLimit:** method, which takes as an argument a window depth limit creating using the **NSBestDepth()** function.

If an NSWindow draws color into its buffer and there's a color screen available, the Window Server automatically promotes the window's depth (up to its limit). This happens whether or not the window is actually on a color screen; similarly, if the user drags a window that displays color from a color to a monochrome screen, it remains at its richer depth. In both cases, the window's depth is greater than the

screen can properly display. Keeping a window's depth at its richest preserves the displayed image, but may produce undesired results such as dithering on a more limited screen, and does causes slight performance reduction when the window buffer is deeper than the screen requires. You can set an NSWindow to keep its depth at the limit of the screen it's on with the **setDynamicDepthLimit:** method. When it's moved to a new screen, a window with a dynamic depth limit is redrawn into the newly adjust buffer. Making a window's depth limit dynamic overrides the limit set using **setDepthLimit:**, and removing the dynamic limit reverts the static limit to the default.

## Window Display and Updating

Display of an NSWindow begins with the drawing performed by its view objects, which accumulates in the window's display buffer or appears immediately on the screen. NSWindows, like NSViews, can be displayed unconditionally or merely marked as needing display, using the **display** and **setViewsNeedDisplay:** methods, respectively. A **displayIfNeeded** message causes the NSWindow's views to display only if they've been marked as needing display. Normally, any time an NSView is marked as needing display the NSWindow makes note of this fact and automatically displays itself shortly after. This automatic display is typically performed on each pass through the event loop, but can be turned off using the **setAutodisplay:** method. If you turn off autodisplay for an NSWindow, you're then responsible for displaying it whenever necessary.

A relating mechanism is that of updating. On each pass through the event loop, the application object invokes its **updateWindows** method, which sends an **update** message to each NSWindow. Subclasses of NSWindow can override this method to examine the state of the application and change their own state or appearance accordingly—enabling or disabling menus, buttons, and other controls based on the object that's selected, for example.

In addition to display on the screen, an NSWindow can print itself in its entirety, just as an NSView does. The **print:** method runs the application's print panel and causes the NSWindow's frame view to print itself. The **fax:** and **dataWithEPSInsideRect:** methods behave similarly. See the NSView class specification for more information on printing.

## Event Handling

As described in the NSResponder class specification, most events coming into an application make their way to a NSWindow in a **sendEvent:** message. A key event is directed at the key window, while a mouse event is directed at whatever window lies under the cursor. If an event affects the NSWindow directly—resizing or moving it, for example—it performs the appropriate operation itself and sends messages to its delegate informing it of its intentions, thus allowing your application to intercede. The window sends other events up its responder chain from the appropriate starting point: the first responder for a key event, the view under the cursor for a mouse event. These events are then typically handled by some view object in the window. See the NSView and NSEvent class specifications for more information on how to intercept and handle events.

The following sections describe aspects of events not directly related to handling individual events. These include changing the key and main windows, which is often handled by views that receive mouse events, but which you must sometimes perform explicitly, such as when opening a new window; changing the first responder by keyboard rather than mouse actions; sharing a single text object for lightweight editing tasks; and running a modal event loop around an entire window rather than a single view object.

#### Changing the Key and Main Windows

Windows already on screen automatically change their status as the key or main window according to the user's actions with the mouse and to how the view clicked handles the mouse event. You can also change the key and main windows programmatically by sending the relevant window object a **makeKeyWindow** or **makeMainWindow** message. This is particularly useful to do when creating a new window. Since this operation is often combined with ordering the window to the front of the screen, NSWindow defines a convenience method, **makeKeyAndOrderFront:**, that performs both operations.

Not all windows are suitable for acting as the key or main window. For example, a window that merely displays information, and contains no objects that need to respond to events or action messages, can completely forgo ever becoming the key window. Similarly, a window that acts as a floating palette of items that are only dragged out by mouse actions never needs to be the key window. Such a window can be defined as a subclass of NSWindow that overrides the methods **canBecomeKeyWindow** and **canBecomeMainWindow** to return NO instead of the default of YES. This prevents it from ever becoming the key or main window. Though NSWindow defines these methods, typically only subclasses of NSPanel refuse to accept key or main window status.

#### Keyboard Interface Control

A window's first responder is often a view object selected by the user clicking on it. For text fields and other view objects (mainly subclasses of NSControl), an NSWindow also allows the user to select the first responder with the keyboard using the Tab and Shift keys. NSView defines the methods for setting up and examining the loop of objects that the user can select in this manner. A view that's the first responder is called the *key view*, and the views that can become the key view in a window are linked together in the window's *key view loop*. You normally set up the key view loop using Interface Builder, establishing connections between the **nextKeyView** outlets of views in the window, and setting the window's **initialFirstResponder** outlet to the view that you want selected when the window is first placed on-screen.

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. **setDefaultButtonCell:** establishes this button cell; you can also set it in Interface Builder by setting a button cell's key equivalent to '\r'. 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. Another default key established by NSWindow is the Escape key, which immediately aborts a modal loop (described below under "Modal Windows").

See the NSResponder class specification for more information on keyboard interface control.

### The Field Editor

Each NSWindow keeps a text object that's shared for light editing tasks. This object, the window's *field editor*, is inserted into the view hierarchy when an object needs to edit some text, and removed when the object is finished.The field editor is used by NSTextFields and other controls, for example, to edit the text that they display. The **fieldEditor:forObject:** method returns an NSWindow's field editor, after asking the delegate for a substitute using **windowWillReturnFieldEditor:toObject:**. You can override the NSWindow method in subclasses or provide a delegate to substitute a different class of text object than the default of NSTextView, thereby customizing text-editing in your application.

### Modal Windows

You can write a modal event loop for a view object to focus events on that object, such as for tracking the mouse. This normally requires you to write your own event loop for the operation in question. You can also make a whole window or panel run in modal fashion, using the application's normal event loop machinery but restricting input to the modal window or panel. This is particularly useful for windows and panels that require the user's attention before an action can proceed: Error messages and warnings are usually presented in modal panels, and operations that require input, such a printing or saving a document, also use modal windows or panels.

OPENSTEP defines two mechanisms for operating a modal window or panel. The first, and simpler, is to invoke NSApplication's **runModalForWindow:** method, which monopolizes events for the NSWindow specified until one of the methods **stopModal**, **abortModal**, or **stopModalWithCode:** is invoked, typically by a button's action method. **stopModal** ends the modal status of the window or panel from within the event loop. It doesn't work if invoked from a method invoked by a timer or by a distributed object, since those mechanisms operate outside of the event loop. To terminate the modal loop in these situations, you can use **abortModal**. **stopModal** is typically invoked when the user clicks the OK button (or equivalent), **abortModal** when the user clicks the Cancel button (or presses the Escape key). These two methods are equivalent to **stopModalWithCode:** with the appropriate argument. See the method descriptions in the NSApplication class specification for more information.

The second mechanism for operating a modal window or panel, called a *modal session*, allows the application to perform a long operation while still bringing events to the window or panel. This is particularly useful for panels that allow the user to cancel or modify an operation. To begin a modal session, invoke NSApplication's **beginModalSessionForWindow:** method, which sets the window up for the session and returns an identifier used for other session-controlling methods. At this point, the application can run in a loop that performs the operation, on each pass sending **runModalSession:** to the application object so that pending events can be dispatched to the modal window. This method returns a code indicating whether the operation should continue, stop, or abort, which is typically established by the methods described above for **runModalForWindow:**. After the loop concludes, you can order the window off-screen and invoke NSApplication's **endModalSession:** method to restore the normal event loop. The method descriptions in the NSApplication class specification include example code illustrating modal sessions.

The normal behavior of a modal window or session is to exclude all other windows and panels from receiving events. This behavior is appropriate for most non-modal windows and panels, but for those that serve as general auxiliary controls, such as menus and the Font Panel, it's overly restrictive. The user must be able to use menu key equivalents (such as for Cut and Paste) and change the font of text in the modal window, and this requires non-modal panels to receiver events. The means for doing this is for the NSWindow subclass to override the **worksWhenModal** method to return YES. Windows that do so can receive mouse and keyboard events even when a modal window is present. If a subclass needs to work when a modal window is present, it should usually be a subclass of NSPanel, not of NSWindow.

Modal windows and sessions provide different levels of control to the application and the user. Modal windows restrict all action to the window itself and any methods invoked from the window. Modal sessions allow the application to continue an operation while accepting input only through the modal session window. Beyond this, you can use distributed objects to perform background operations in a separate thread, while allowing the user to perform other actions with any part of the application. The background thread can communicate with the main thread, allowing the application to display the status of the operation in a non-modal panel, perhaps include controls to stop or affect the operations as it occurs. Note that because the Application Kit isn't thread-safe, the background thread should communicate with a designated object in the main thread that in turn interacts with the Application Kit.

## Notifications and the NSWindow's Delegate

NSWindow offers observers a rich set of notifications, which it broadcasts on such occurrences as gaining or losing key or main window status, miniaturizing, moving or resizing, becoming exposed, and closing. Each notification is matched to a delegate method, so an NSWindow's delegate is automatically registered for all notifications that it has methods for. NSWindow also offers its delegate a few other methods, such as **windowShouldClose:**, which requests approval to close, **windowWillResize:toSize:**, which allows the delegate to constrain the NSWindow's size, and **windowWillReturnFieldEditor:toObject:**, which gives the delegate a chance to modify the field editor or substitute a different editor. See the individual notification and delegate method descriptions at the end of this specification for more information.

## Other Features

NSWindow defines a number of methods to assist its view objects in certain operations that may extend in scope beyond a single view or even outside the window containing them. First of these operations is image dragging. Although most dragging operations are initiated by and occur between view objects, NSWindow also defines an image-dragging method, **dragImage:at:offset:event:pasteboard:source:slideBack:**. An NSWindow can also serve as the destination for dragging operations, registering the types it accepts with **registerForDraggedTypes:** and **unregisterForDraggedTypes**. See the descriptions of these same methods in the NSView class specification for more information.

NSViews also handle definition of cursor rectangles—areas where the cursor image changes when the mouse enters them. NSWindow can disable and reenable all of its cursor rectangles with the **disableCursorRects** and **enableCursorRects** methods. You can determine whether they're enabled using

**areCursorRectsEnabled**. To reset the cursor rectangles for a particular NSView, invoke the **invalidateCursorRectsForView:** method, and to reset them all, use **resetCursorRects**.

Finally, to support transitory drawing by NSViews, NSWindow declares methods that temporarily cache a portion of its raster image so that it can be restored later. This feature is useful for situations where highly dynamic drawing must be done over the otherwise static image of the window. For example, in a drawing program where the user drags lines and other shapes directly onto a canvas, it's more efficient to restore the window's cached image and draw anew over that than to have all of the view objects send PostScript instructions to the Window Server. For more information, see "Transitory Drawing" in the NSView class specification, and the individual method descriptions of **cacheImageInRect:**, **restoreCachedImage**, and **discardCachedImage** in this class specification.

## Method Types

| | |
|---|---|
| Creating instances | – initWithContentRect:styleMask:backing:defer:<br>– initWithContentRect:styleMask:backing:defer:screen: |
| Calculating layout | + contentRectForFrameRect:styleMask:<br>+ frameRectForContentRect:styleMask:<br>+ minFrameWidthWithTitle:styleMask: |
| Converting coordinates | – convertBaseToScreen:<br>– convertScreenToBase: |
| Moving and resizing | – setFrame:display:<br>– frame<br>– setFrameOrigin:<br>– setFrameTopLeftPoint:<br>– setContentSize:<br>– cascadeTopLeftFromPoint:<br>– center<br>– resizeFlags |
| Constraining window size | – maxSize<br>– minSize<br>– setMaxSize:<br>– setMinSize:<br>– setAspectRatio:<br>– aspectRatio<br>– setResizeIncrements:<br>– resizeIncrements<br>– constrainFrameRect:toScreen: |

| | |
|---|---|
| Saving the frame to user defaults | + removeFrameUsingName:<br>– saveFrameUsingName:<br>– setFrameUsingName:<br>– setFrameAutosaveName:<br>– frameAutosaveName<br>– setFrameFromString:<br>– stringWithSavedFrame |
| Ordering windows | – orderBack:<br>– orderFront:<br>– orderFrontRegardless<br>– orderOut:<br>– orderWindow:relativeTo:<br>– setLevel:<br>– level<br>– isVisible |
| Making key and main windows | – becomeKeyWindow<br>– canBecomeKeyWindow<br>– isKeyWindow<br>– makeKeyAndOrderFront:<br>– makeKeyWindow<br>– resignKeyWindow<br>– becomeMainWindow<br>– canBecomeMainWindow<br>– isMainWindow<br>– makeMainWindow<br>– resignMainWindow |
| Display and drawing | – display<br>– displayIfNeeded<br>– setViewsNeedDisplay:<br>– viewsNeedDisplay<br>– useOptimizedDrawing:<br>– setAutodisplay:<br>– isAutodisplay<br>– update |
| Flushing graphics | – flushWindow<br>– flushWindowIfNeeded<br>– enableFlushWindow<br>– disableFlushWindow<br>– isFlushWindowDisabled |

| | |
|---|---|
| Bracketing temporary drawing | – cacheImageInRect:<br>– restoreCachedImage<br>– discardCachedImage |
| Window Server information | – windowNumber<br>– gState<br>– deviceDescription<br>– setBackingType:<br>– backingType<br>– setOneShot:<br>– isOneShot<br>+ defaultDepthLimit<br>– setDepthLimit:<br>– depthLimit<br>– setDynamicDepthLimit:<br>– hasDynamicDepthLimit<br>– canStoreColor |
| Screen information | – deepestScreen<br>– screen |
| Working with the responder chain | – makeFirstResponder:<br>– firstResponder |
| Event handling | – currentEvent<br>– nextEventMatchingMask:<br>– nextEventMatchingMask:untilDate:inMode:dequeue:<br>– discardEventsMatchingMask:beforeEvent:<br>– postEvent:atStart:<br>– sendEvent:<br>– tryToPerform:with:<br>– keyDown:<br>– mouseLocationOutsideOfEventStream<br>– setAcceptsMouseMovedEvents:<br>– acceptsMouseMovedEvents |
| Working with the field editor | – fieldEditor:forObject:<br>– endEditingFor: |
| Keyboard interface control | – setInitialFirstResponder:<br>– initialFirstResponder<br>– selectKeyViewFollowingView:<br>– selectKeyViewPrecedingView:<br>– selectNextKeyView:<br>– selectPreviousKeyView:<br>– keyViewSelectionDirection |

| | |
|---|---|
| Setting the title and filename | – setTitle:<br>– setTitleWithRepresentedFilename:<br>– title<br>– setRepresentedFilename:<br>– representedFilename |
| Marking a window edited | – setDocumentEdited:<br>– isDocumentEdited |
| Closing the window | – close<br>– performClose:<br>– setReleasedWhenClosed:<br>– isReleasedWhenClosed |
| Miniaturizing and miniwindows | – miniaturize:<br>– performMiniaturize:<br>– deminiaturize:<br>– isMiniaturized<br>– setMiniwindowImage:<br>– miniwindowImage<br>– setMiniwindowTitle:<br>– miniwindowTitle |
| Working with menus | + menuChanged: |
| Working with the Windows menu | – setExcludedFromWindowsMenu:<br>– isExcludedFromWindowsMenu |
| Working with cursor rectangles | – areCursorRectsEnabled<br>– enableCursorRects<br>– disableCursorRects<br>– discardCursorRects<br>– invalidateCursorRectsForView:<br>– resetCursorRects |
| Dragging | – dragImage:at:offset:event:pasteboard:source:slideBack:<br>– registerForDraggedTypes:<br>– unregisterDraggedTypes |
| Controlling behavior | – setHidesOnDeactivate:<br>– hidesOnDeactivate<br>– worksWhenModal |
| Setting the content view | – setContentView:<br>– contentView |
| Setting background color | – setBackgroundColor:<br>– backgroundColor |
| Getting the style mask | – styleMask |

| | |
|---|---|
| Working with Services | – validRequestorForSendType:returnType: |
| Printing and faxing | – print:<br>– dataWithEPSInsideRect:<br>– fax: |
| Getting the Microsoft Windows handle | – windowHandle |
| Setting the delegate | – setDelegate:<br>– delegate |

## Class Methods

### contentRectForFrameRect:styleMask:

+ (NSRect)**contentRectForFrameRect:**(NSRect)*frameRect* **styleMask:**(unsigned int)*aStyle*

Returns the content rectangle used by an NSWindow with a frame rectangle of *frameRect* and a style mask of *aStyle*. Both *frameRect* and the returned content rectangle are expressed in screen coordinates. See the **initWithContentRect:styleMask:backing:defer:** method description for a list of style mask values.

**See also:** + **frameRectForContentRect:styleMask:**

### defaultDepthLimit

+ (NSWindowDepth)**defaultDepthLimit**

Returns the default depth limit for instances of NSWindow. This is the smaller of:

- The depth of the deepest display device available to the Window Server
- The depth set for the application by the NSWindowDepthLimit parameter

The value returned can be examined with the Application Kit functions **NSPlanarFromDepth()**, **NSColorSpaceFromDepth()**, **NSBitsPerSampleFromDepth()**, **NSBitsPerPixelFromDepth()**.

**See also:** – **setDepthLimit:**, – **setDynamicDepthLimit:**, – **canStoreColor**

### frameRectForContentRect:styleMask:

+ (NSRect)**frameRectForContentRect:**(NSRect)*contentRect* **styleMask:**(unsigned int)*aStyle*

Returns the frame rectangle used by an NSWindow with a content rectangle of *contentRect* and a style mask of *aStyle*. Both *contentRect* and the returned frame rectangle are expressed in screen coordinates. See the **initWithContentRect:styleMask:backing:defer:** method description for a list of style mask values.

**See also:** + **contentRectForContentRect:styleMask:**

### menuChanged:

+ (void)**menuChanged:**(NSMenu *)*aMenu*

On Microsoft Windows, locates all objects inheriting from NSWindow that use *aMenu* and causes them to update their state and redisplay the menu. On Mach, this method does nothing.

**See also:** – **menu** (NSResponder)

### minFrameWidthWithTitle:styleMask:

+ (float)**minFrameWidthWithTitle:**(NSString *)*aTitle* **styleMask:**(unsigned int)*aStyle*

Returns the minimum width that an NSWindow's frame rectangle must have for it to display all of *aTitle*, given *aStyle* as its style mask. See the **initWithContentRect:styleMask:backing:defer:** method description for a list of acceptable style mask values.

### removeFrameUsingName:

+ (void)**removeFrameUsingName:**(NSString *)*name*

Removes the frame data stored under *name* from the application's user defaults.

**See also:** – **setFrameUsingName:**, – **setFrameAutosaveName:**

## Instance Methods

### acceptsMouseMovedEvents

– (BOOL)**acceptsMouseMovedEvents**

Returns YES if the receiver accepts and distributes mouse-moved events, NO if it doesn't. NSWindows by default don't accept mouse-moved events.

**See also:** – **setAcceptsMouseMovedEvents:**

### areCursorRectsEnabled

– (BOOL)**areCursorRectsEnabled**

Returns YES if the receiver's cursor rectangles are enabled, NO if they're not.

**See also:** – **setCursorRectsEnabled:**, – **addCursorRect:cursor:** (NSView)

## aspectRatio

– (NSSize)**aspectRatio**

Returns the receiver's size aspect ratio. The size of the receiver's frame rectangle is constrained to integral multiples of this ratio when the user resizes it. You can set an NSWindow's size to any ratio programmatically.

**See also:** – **resizeIncrements**, – **setAspectRatio:**, – **setFrame:display:**

## backgroundColor

– (NSColor *)**backgroundColor**

Returns the color of the receiver's background.

**See also:** – **setBackgroundColor:**

## backingType

– (NSBackingStoreType)**backingType**

Returns the receiver's backing store type as one of the following constants:

NSBackingStoreBuffered
NSBackingStoreRetained
NSBackingStoreNonretained

**See also:** – **setBackingType:**

## becomeKeyWindow

– (void)**becomeKeyWindow**

Invoked automatically to inform the receiver that it has become the key window; never invoke this method directly. This method reestablishes the receiver's first responder, sends **becomeKeyWindow** to that object if it responds, and posts an NSWindowDidBecomeKeyNotification to the default notification center.

**See also:** – **makeKeyWindow**, – **makeKeyAndOrderFront:**, – **becomeMainWindow**

## becomeMainWindow

– (void)**becomeMainWindow**

Invoked automatically to inform the receiver that it has become the main window; never invoke this method directly. This method posts an NSWindowDidBecomeMainNotification to the default notification center.

**See also:** – **makeMainWindow**, – **becomeKeyWindow**

## cacheImageInRect:

– (void)**cacheImageInRect:**(NSRect)*aRect*

Stores the receiver's raster image from *aRect*, which is expressed in the receiver's base coordinate system. This allows the receiver to perform temporary drawing, such as a band around the selection as the user drags the mouse, and to quickly restore the previous image by invoking **restoreCachedImage** and **flushWindowIfNeeded**. The next time the window displays, it discards its cached image rectangles. You can also explicitly use **discardCachedImage** to free the memory occupied by cached image rectangles.

**See also:** – **display**

## canBecomeKeyWindow

– (BOOL)**canBecomeKeyWindow**

Returns YES if the receiver is able to be the key window, and NO if it can't. Attempts to make the receiver the key window are abandoned if this method returns NO. NSWindow's implementation returns YES if the receiver has a title bar or a resize bar/border, NO otherwise.

**See also:** – **isKeyWindow**, – **makeKeyWindow**

## canBecomeMainWindow

– (BOOL)**canBecomeMainWindow**

Returns YES if the receiver is able to be the main window, and NO if it can't. Attempts to make the receiver the main window are abandoned if this method returns NO. NSWindow's implementation returns YES if the receiver is visible, is *not* an NSPanel, and has a title bar or a resize bar/border. Otherwise it returns NO.

**See also:** – **isMainWindow**, – **makeMainWindow**

### canStoreColor

– (BOOL)**canStoreColor**

Returns YES if the receiver has a depth limit that allows it to store color values, and NO if it doesn't.

**See also:** – **depthLimit**, – **shouldDrawColor** (NSView)

### cascadeTopLeftFromPoint:

– (NSPoint)**cascadeTopLeftFromPoint:**(NSPoint)*topLeftPoint*

Returns a point shifted from *topLeftPoint* that can be used to place the receiver in a cascade relative to another NSWindow positioned at *topLeftPoint*, so that the title bars of both NSWindows are fully visible. Both points are expressed in screen coordinates.

**See also:** – **setFrameTopLeftPoint:**

### center

– (void)**center**

Sets the receiver's location to the center of the screen: The receiver is placed dead-center horizontally and somewhat above center vertically. Such a placement is considered to carry a certain visual immediacy and importance. This method doesn't put the receiver on-screen, however, use **makeKeyAndOrderFront:** to do that.

You typically use this method to place a NSWindow—most likely an attention panel—where the user can't miss it. This method is invoked automatically when an NSPanel is placed on the screen by NSApplication's **runModalForWindow:** method.

### close

– (void)**close**

Removes the receiver from the screen. If the receiver is set to be released when it's closed (the default), a **release** message is sent to the object after the current event is completed.

This method doesn't result in any notification that the receiver is closing, as **performClose:** and closing due to user action do. Use **performClose:** if you want to be sure that observers and the delegate will be notified of the receiver's closing.

**See also:** – **setReleasedWhenClosed:**

### constrainFrameRect:toScreen:

– (NSRect)**constrainFrameRect:**(NSRect)*frameRect* **toScreen:**(NSScreen *)*aScreen*

Modifies and returns *frameRect* so that its top edge lies on *aScreen*. If the receiver is resizable, the rectangle's height is adjusted to bring the bottom edge onto the screen as well. The rectangle's width and horizontal location are unaffected. You shouldn't need to invoke this method yourself; it's invoked automatically (and the modified frame is used to locate and set the size of the receiver) whenever a titled NSWindow is placed on-screen and whenever its size is changed.

Subclasses can override this method to prevent their instances from being constrained, or to constrain them differently.

### contentView

– (id)**contentView**

Returns the receivers's content view, the highest accessible NSView object in the receiver's view hierarchy.

**See also:** – **setContentView:**

### convertBaseToScreen:

– (NSPoint)**convertBaseToScreen:**(NSPoint)*aPoint*

Converts *aPoint* from the receiver's base coordinate system to the screen coordinate system. Returns the converted point.

**See also:** – **convertScreenToBase:**, – **convertPoint:toView:** (NSView)

### convertScreenToBase:

– (NSPoint)**convertScreenToBase:**(NSPoint)*aPoint*

Converts *aPoint* from the screen coordinate system to the receiver's base coordinate system. Returns the converted point.

**See also:** – **convertBaseToScreen:**, – **convertPoint:fromView:** (NSView)

### currentEvent

– (NSEvent *)**currentEvent**

Returns the event currently being processed by the application, by invoking NSApp's **currentEvent** method.

### dataWithEPSInsideRect:

– (NSData *)**dataWithEPSInsideRect:**(NSRect)*aRect*

Returns EPS data that draws the region of the receiver within *aRect* (which is expressed in the receiver's base coordinate system). This data can be placed on an NSPasteboard, written to a file, or used to create an NSImage object.

**See also:** – **dataWithEPSInsideRect:** (NSView), – **writeEPSInsideRect:toPasteboard:** (NSView)

### deepestScreen

– (NSScreen *)**deepestScreen**

Returns the deepest screen that the receiver is on (it may be split over several screens), or **nil** if the receiver is off-screen.

**See also:** – **screen**

### defaultButtonCell:

– (NSButtonCell *)**defaultButtonCell**

Returns the button cell that performs as if clicked when the NSWindow receives a Return (or Enter) key event. This cell draws itself as if it were the focal element for keyboard interface control, unless another button cell is focused on, in which case the default button cell temporarily draws itself as normal and disables its key equivalent.

The window receives a Return key event if no responder in its responder chain claims it, or if the user presses the Control key along with the Return key.

**See also:** – **setDefaultButtonCell:**, – **disableKeyEquivalentForDefaultButtonCell**, – **enableKeyEquivalentForDefaultButtonCell**

### delegate

– (id)**delegate**

Returns the receiver's delegate, or **nil** if it doesn't have one.

**See also:** – **setDelegate:**

## deminiaturize:

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

Deminiaturizes the receiver. You rarely need to invoke this method; it's invoked automatically when an NSWindow is deminiaturized by the user.

**See also:** – **miniaturize:**, – **styleMask**

## depthLimit

– (NSWindowDepth)**depthLimit**

Returns the depth limit of the receiver. The value returned can be examined with the Application Kit functions **NSPlanarFromDepth()**, **NSColorSpaceFromDepth()**, **NSBitsPerSampleFromDepth()**, **NSBitsPerPixelFromDepth()**.

**See also:** + **defaultDepthLimit**, – **setDepthLimit:**, – **setDynamicDepthLimit:**

## deviceDescription

– (NSDictionary *)**deviceDescription**

Returns a dictionary containing information about the receiver's resolution, color depth, and so on. This information is useful for tuning images and colors to the window's display capabilities. The contents of the dictionary are:

| Dictionary Key | Value |
|---|---|
| NSDeviceResolution | An NSValue containing an NSSize that describe the receiver's raster resolution in dots per inch (dpi). |
| NSDeviceColorSpaceName | An NSString giving the name of the receiver's color space. See the Application Kit Types and Constants for a list of possible values. |
| NSDeviceBitsPerSample | An NSNumber containing an integer that gives the bit depth of the receiver's raster image (2-bit, 8-bit, etc.). |
| NSDeviceIsScreen | "YES", indicating that the receiver displays on the screen. |
| NSDeviceSize | An NSValue containing an NSSize that gives the size of the receiver's frame rectangle. |

**See also:** – **deviceDescription** (NSScreen), – **bestRepresentationForDevice:** (NSImage), – **colorUsingColorSpaceName:** (NSColor)

## disableCursorRects

– (void)**disableCursorRects**

Disables all cursor rectangle management within the receiver. Use this method when you need to do some special cursor manipulation and you don't want the Application Kit interfering.

**See also:** – **enableCursorRects**

## disableFlushWindow

– (void)**disableFlushWindow**

Disables the **flushWindow** method for the receiver. If the receiver is buffered, this prevents drawing from being automatically flushed by NSView's **display...** methods from the receiver's backing store to the screen. This permits several NSViews to be displayed before the results are shown to the user.

Flushing should be disabled only temporarily, while the NSWindow's display is being updated. Each **disableFlushWindow** message must be paired with a subsequent **enableFlushWindow** message. Invocations of these methods can be nested; flushing isn't reenabled until the last (unnested) **reenableFlushWindow** message is sent.

## disableKeyEquivalentForDefaultButtonCell

– (void)**disableKeyEquivalentForDefaultButtonCell**

Disables the default button cell's key equivalent, so that it doesn't perform a click when the user presses Return (or Enter). See the method description for **defaultButtonCell** for more information.

**See also:** – **enableKeyEquivalentForDefaultButtonCell**

## discardCachedImage

– (void)**discardCachedImage**

Discards all of the receiver's cached image rectangles. An NSWindow automatically discards its cached image rectangles when it displays.

**See also:** – **cacheImageInRect:**, – **restoreCachedImage**, – **display**

## discardCursorRects

– (void)**discardCursorRects**

Invalidates all cursor rectangles in the receiver. This method is invoked by **resetCursorRects** to clear out existing cursor rectangles before resetting them. You shouldn't invoke it in the code you write, but might want to override it to change its behavior.

**See also:** – **resetCursorRects**

## discardEventsMatchingMask:beforeEvent:

– (void)**discardEventsMatchingMask:**(unsigned int)*mask* **beforeEvent:**(NSEvent *)*lastEvent*

Forwards the message to the NSApplication object.

## display

– (void)**display**

Passes a **display** message down the receiver's view hierarchy, thus redrawing all NSViews within the receiver, including the frame view which draws the border, title bar, and other peripheral elements.

You rarely need to invoke this method. NSWindows normally record which of their NSViews need display and display them automatically on each pass through the event loop.

**See also:** – **display** (NSView), – **displayIfNeeded**, – **isAutodisplay**

## displayIfNeeded

– (void)**displayIfNeeded**

Passes a **displayIfNeeded** message down the receiver's view hierarchy, thus redrawing all NSViews that need to be displayed, including the frame view which draws the border, title bar, and other peripheral elements. This method is useful when you want to modify some number of NSViews, and then display only the ones that were modified.

You rarely need to invoke this method. NSWindows normally record which of their NSViews need display and display them automatically on each pass through the event loop.

**See also:** – **display**, – **displayIfNeeded** (NSView), – **setNeedsDisplay:** (NSView), – **isAutodisplay**

### dragImage:at:offset:event:pasteboard:source:slideBack:

– (void)**dragImage:**(NSImage *)*anImage*
**at:**(NSPoint)*aPoint*
**offset:**(NSSize)*initialOffset*
**event:**(NSEvent *)*theEvent*
**pasteboard:**(NSPasteboard *)*pboard*
**source:**(id)*sourceObject*
**slideBack:**(BOOL)*flag*

Begins a dragging session. This method is essentially the same as NSView's method of the same name, except that *aPoint* is given in the NSWindow's base coordinate system. This method should be invoked only from within an NSView's implementation of the **mouseDown:** method. See the description of this method in the NSView class specification for more information.

### enableCursorRects

– (void)**enableCursorRects**

Reenables cursor rectangle management within the receiver after a **disableCursorRects** message.

### enableFlushWindow

– (void)**enableFlushWindow**

Reenables the **flushWindow** method for the receiver after it was disabled through a previous **disableFlushWindow** message.

### enableKeyEquivalentForDefaultButtonCell

– (void)**enableKeyEquivalentForDefaultButtonCell**

Reenables the default button cell's key equivalent, so that it performs a click when the user presses Return (or Enter). See the method description for **defaultButtonCell** for more information.

**See also:** – **disableKeyEquivalentForDefaultButtonCell**

### endEditingFor:

– (void)**endEditingFor:**(id)*anObject*

Forces the field editor, which *anObject* is assumed to be using, to give up its first responder status, and prepares it for its next assignment. If the field editor is the first responder, it's made to resign that status even if its **resignFirstResponder** method returns NO. This forces the field editor to send a

**textDidEndEditing:** message to its delegate. The field editor is then removed from the view hierarchy, its delegate is set to **nil**, and it's emptied of any text it may contain.

This method is typically invoked by the object using the field editor when it's finished. Other objects normally change the first responder by simply using **makeFirstResponder:**, which allows a field editor or other object to retain its first responder status if, for example, the user has entered an invalid value. **endEditingFor:** should be used only as a last resort if the field editor refuses to resign first responder status. Even in this case, you should always allow the field editor a chance to validate its text and take whatever other action it needs first. You can do this by first trying to make the NSWindow the first responder:

```
if ([myWindow makeFirstResponder:myWindow]) {
    /* All fields are now valid; it's safe to use fieldEditor:forObject:
     * to claim the field editor. */
}
else {
    /* Force first responder to resign. */
    [myWindow endEditingFor:nil];
}
```

**See also:** – **fieldEditor:forObject:**, – **windowWillReturnFieldEditor:toObject:**

## fax:

– (void)**fax:**(id)*sender*

Runs the Fax panel, and if the user chooses an option other than canceling, prints the receiver (its frame view and all subviews) to a fax modem.

**See also:** – **print:**

## fieldEditor:forObject:

– (NSText *)**fieldEditor:**(BOOL)*createFlag* **forObject:**(id)*anObject*

Returns the receiver's field editor, creating it if needed and if *createFlag* is YES. Returns **nil** if *createFlag* is NO and the field editor doesn't exist. *anObject* is used to allow the receiver's delegate to substitute another object in place of the field editor, as described below. The field editor may be in use by some view object, so be sure to properly dissociate it from that object before actually using it yourself (the appropriate way to do this is illustrated in the description of **endEditingFor:**). Once you retrieve the field editor, you can insert it into the view hierarchy and set a delegate to interpret text events, have it perform whatever editing is needed. Then, when it sends a **textDidEndEditing:** message to the delegate, you can get its text to display or store, and remove the field editor using **endEditingFor:**.

The field editor is provided as a convenience and can be used however your application sees fit. Typically, the field editor is used by simple text-bearing objects—for example, an NSTextField object uses its window's field editor to display and manipulate text. The field editor can be shared by any number of

objects and so its state may be constantly changing. Therefore, it shouldn't be used to display text that demands sophisticated layout (for this you should create a dedicated NSText object).

A freshly created NSWindow doesn't have a field editor. After a field editor has been created for an NSWindow, the *createFlag* argument is ignored. By passing NO for *createFlag* and testing the return value, however, you can predicate an action on the existence of the field editor.

The receiver's delegate can substitute a custom editor in place of the NSWindow's field editor by implementing **windowWillReturnFieldEditor:toObject:**. The receiver sends this message to its delegate with itself and *anObject* as the arguments, and if the return value is non-**nil** the NSWindow returns that object instead of its field editor. However, note the following:

- If the NSWindow's delegate is identical to *anObject*, **windowWillReturnFieldEditor:toObject:** isn't sent.
- The object returned by the delegate method, though it may become first responder, does *not* become the NSWindow's field editor. Other objects continue to use the NSWindow's established field editor.

## firstResponder

– (NSResponder *)**firstResponder**

Returns the receiver's first responder.

**See also:** – **makeFirstResponder:**, – **acceptsFirstResponder** (NSResponder)

## flushWindow

– (void)**flushWindow**

Flushes the receiver's off-screen buffer to the screen, if the receiver is buffered and flushing is enabled. Does nothing for other display devices, such as a printer. This method is automatically invoked by NSWindow's and NSView's **display...** methods.

**See also:** – **flushWindowIfNeeded**, – **display...** (NSView), – **disableFlushWindow**, – **enableFlushWindow**

## flushWindowIfNeeded

– (void)**flushWindowIfNeeded**

Flushes the receiver's off-screen buffer to the screen, if flushing is enabled and if the last **flushWindow** message had no effect because flushing was disabled. To avoid unnecessary flushing, use this method rather than **flushWindow** to flush an NSWindow after flushing has been reenabled.

**See also:** – **flushWindow**, – **disableFlushWindow**, – **enableFlushWindow**

## frame

– (NSRect)**frame**

Returns the receiver's frame rectangle. The frame rectangle is always reckoned in the screen coordinate system.

**See also:** – **screen**, – **deepestScreen**

## frameAutosaveName

– (NSString *)**frameAutosaveName**

Returns the name used to automatically save the receiver's frame rectangle data in the defaults system, as set through **setFrameAutosaveName:**. If the receiver has an autosave name, its frame data is written whenever the frame rectangle changes.

**See also:** – **setFrameUsingName:**

## gState

– (int)**gState**

Returns the PostScript graphics state object associated with the receiver. This graphics state is used by default for all NSViews in the receiver's view hierarchy, but individual NSViews can be made to use their own with the NSView method **allocateGState**.

## hasDynamicDepthLimit

– (BOOL)**hasDynamicDepthLimit**

Returns YES if the receiver's depth limit can change to match the depth of the screen it's on, NO if it can't.

**See also:** – **setDynamicDepthLimit:**

## hidesOnDeactivate

– (BOOL)**hidesOnDeactivate**

Returns YES if the receiver is removed from the screen when its application is deactivated, and NO if it remains on-screen.

**See also:** – **setHidesOnDeactivate:**

## initialFirstResponder

– (NSView *)**initialFirstResponder**

Returns the NSView that's made first responder the first time the receiver is placed on-screen.

**See also:** – **setInitialFirstResponder:**, – **setNextKeyView:**

## initWithContentRect:styleMask:backing:defer:

– (id)**initWithContentRect:**(NSRect)*contentRect*
**styleMask:**(unsigned int)*styleMask*
**backing:**(NSBackingStoreType)*backingType*
**defer:**(BOOL)*flag*

Initializes the receiver, a newly allocated NSWindow object, and returns **self**. This method is the designated initializer for the NSWindow class.

*contentRect* specifies the location and size of the NSWindow's content area in screen coordinates. Note that the Window Server limits window position coordinates to ±16,000 and sizes to 10,000.

*styleMask* specifies the receiver's style. It can either be NSBorderlessWindowMask, or it can contain any of the following options, combined using the C bitwise OR operator:

| Option | Meaning |
|---|---|
| NSTitledWindowMask | The NSWindow displays a title bar. |
| NSClosableWindowMask | The NSWindow displays a close button. |
| NSMiniaturizableWindowMask | The NSWindow displays a miniaturize button. |
| NSResizableWindowMask | The NSWindow displays a resize bar or border. |

Borderless windows display none of the usual peripheral elements, and are generally useful only for display or caching purposes; you should normally not need to create them. Also, note that an NSWindow's style mask should include NSTitledWindowMask if it includes any of the others.

*backingType* specifies how the drawing done in the receiver is buffered by the object's window device:

NSBackingStoreBuffered
NSBackingStoreRetained
NSBackingStoreNonretained

*flag* determines whether the Window Server creates a window device for the new object immediately. If *flag* is YES, it defers creating the window until the receiver is ordered on-screen. All display messages sent to the NSWindow or its NSViews are postponed until the window is created, just before it's moved on-screen. Deferring the creation of the window improves launch time and minimizes the virtual memory load on the Server.

The new NSWindow creates an instance of NSView to be its default content view. You can replace it with your own object by using the **setContentView:** method.

**See also:** – **orderFront:**, – **setTitle:**, – **setOneShot:**,
– **initWithContentRect:styleMask:backing:defer:screen:**

## initWithContentRect:styleMask:backing:defer:screen:

– (id)**initWithContentRect:**(NSRect)*contentRect*
**styleMask:**(unsigned int)*styleMask*
**backing:**(NSBackingStoreType)*bufferingType*
**defer:**(BOOL)*flag*
**screen:**(NSScreen *)*aScreen*

Initializes a newly allocated NSWindow object and returns **self**. This method is equivalent to **initWithContentRect:styleMask:backing:defer:screen:**, except that the content rectangle is specified relative to the lower left corner of *aScreen*.

If *aScreen* is **nil**, the content rectangle is interpreted relative to the lower left corner of the main screen. The main screen is the one that contains the current key window, or, if there is no key window, the one that contains the main menu. If there's neither a key window nor a main menu (if there's no active application), the main screen is the one where the origin of the screen coordinate system is located.

**See also:** – **orderFront:**, – **setTitle:**, – **setOneShot:**, – **initWithContentRect:styleMask:backing:defer:**

## invalidateCursorRectsForView:

– (void)**invalidateCursorRectsForView:**(NSView *)*aView*

Marks as invalid the cursor rectangles of *aView*, an NSView in the receiver's view hierarchy, so that they'll be set up again when the receiver becomes key (or immediately if the receiver is key).

**See also:** – **resetCursorRects**, – **resetCursorRects** (NSView)

## isAutodisplay

– (BOOL)**isAutodisplay**

Returns YES if the receiver automatically displays its views that are marked as needing it, NO if it doesn't. Automatic display typically occurs on each pass through the event loop.

**See also:** – **setAutodisplay:**, – **displayIfNeeded**, – **setNeedsDisplay:** (NSView)

### isDocumentEdited

– (BOOL)**isDocumentEdited**

Returns YES or NO according to the argument supplied with the last **setDocumentEdited:** message.

### isExcludedFromWindowsMenu

– (BOOL)**isExcludedFromWindowsMenu**

Returns YES if the receiver's title is omitted from the application's Windows menu, and NO if it is listed.

**See also:** – **setExcludedFromWindowsMenu:**

### isFlushWindowDisabled

– (BOOL)**isFlushWindowDisabled**

Returns YES if the receiver's flushing ability has been disabled; otherwise returns NO.

**See also:** – **disableFlushWindow**, – **enableFlushWindow**

### isKeyWindow

– (BOOL)**isKeyWindow**

Returns YES if the receiver is the key window for the application, and NO if it isn't.

**See also:** – **isMainWindow**, – **makeKeyWindow**

### isMainWindow

– (BOOL)**isMainWindow**

Returns YES if the receiver is the main window for the application, and NO if it isn't.

**See also:** – **isKeyWindow**, – **makeMainWindow**

### isMiniaturized

– (BOOL)**isMiniaturized**

Returns YES if the receiver has been miniaturized, NO if it hasn't. A miniaturized window is removed from the screen and replaced by a miniwindow, icon, or button that represents it, called the *counterpart* (the particular form depends on the platform).

**See also:** – **miniaturize:**

### isOneShot

– (BOOL)**isOneShot**

Returns YES if the PostScript window device that the receiver manages is freed when it's removed from the screen list, and NO if not. The default is NO.

**See also:** – **setOneShot:**

### isReleasedWhenClosed

– (BOOL)**isReleasedWhenClosed**

Returns YES if the receiver is automatically released after being closed, NO if it's simply removed from the screen. The default for NSWindow is YES; the default for NSPanel is NO.

**See also:** – **setReleasedWhenClosed:**

### isVisible

– (BOOL)**isVisible**

Returns YES if the receiver is on-screen (even if it's obscured by other windows).

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

### keyDown:

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

Handles a keyboard event that may need to be interpreted as changing the key view or triggering a mnemonic.

**See also:** – **selectNextKeyView:**, – **nextKeyView** (NSView), – **performMnemonic:** (NSView)

## keyViewSelectionDirection

– (NSSelectionDirection)**keyViewSelectionDirection**

Returns the direction that the receiver is currently using to change the key view, one of:

| Value | Meaning |
|---|---|
| NSDirectSelection | The receiver isn't traversing the key view loop. |
| NSSelectingNext | The receiver is proceeding to the next valid key view. |
| NSSelectingPrevious | The receiver is proceeding to the previous valid key view. |

**See also:** – **selectNextKeyView:**, – **selectPreviousKeyView:**

## level

– (int)**level**

Returns the level of the receiver as set using **setLevel:**. See that method description for a list of possible values.

## makeFirstResponder:

– (BOOL)**makeFirstResponder:**(NSResponder *)*aResponder*

Attempts to make *aResponder* the first responder for the receiver. If *aResponder* isn't already the first responder, this method first sends a **resignFirstResponder** message to the object that is. If that object refuses to resign, it remains the first responder and this method immediately returns NO. If it returns YES, this methods sends a **becomeFirstResponder** message to *aResponder*. If *aResponder* accepts first responder status, this method returns YES. If it refuses, this method returns NO, and the NSWindow becomes first responder.

The Application Kit uses this method to alter the first responder in response to mouse-down events; you can also use it to explicitly set the first responder from within your program. *aResponder* is typically an NSView in the receiver's view hierarchy.

**See also:** – **becomeFirstResponder** (NSResponder), – **resignFirstResponder** (NSResponder)

## makeKeyAndOrderFront:

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

Moves the receiver to the front of the screen list, within its level, and makes it the key window.

**See also:** – **orderFront:**, – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **setlevel:**

### makeKeyWindow

– (void)**makeKeyWindow**

Makes the receiver the key window.

**See also:** – **makeMainWindow**, – **becomeKeyWindow**, – **isKeyWindow**

### makeMainWindow

– (void)**makeMainWindow**

Makes the receiver the main window.

**See also:** – **makeKeyWindow**, – **becomeMainWindow**, – **isMainWindow**

### maxSize

– (NSSize)**maxSize**

Returns the maximum size to which the receiver's frame can be sized either by the user or by the **setFrame...** methods other than **setFrame:display:**.

**See also:** – **setMaxSize:**, – **minSize**, – **aspectRatio**, – **resizeIncrements**

### miniaturize:

– (void)**miniaturize:**(id)*sender*

Removes the receiver from the screen list and displays its counterpart in the appropriate location.

**See also:** – **deminiaturize:**

### miniwindowImage

– (NSImage *)**miniwindowImage**

Returns the image that's displayed in the receiver's miniwindow.

**See also:** – **setMiniwindowImage:**, – **miniwindowTitle**

### miniwindowTitle

– (NSString *)**miniwindowTitle**

Returns the title that's displayed in the receiver's miniwindow.

**See also:** – **setMiniwindowTitle:**, – **miniwindowImage**

### minSize

– (NSSize)**minSize**

Returns the minimum size to which the receiver's frame can be sized either by the user or by the **setFrame...** methods other than **setFrame:display:**.

**See also:** – **setMinSize:**, – **maxSize**, – **aspectRatio**, – **resizeIncrements**

### mouseLocationOutsideOfEventStream

– (NSPoint)**mouseLocationOutsideOfEventStream**

Returns the current location of the mouse reckoned in the receiver's base coordinate system, regardless of the current event being handled or of any events pending.

**See also:** – **currentEvent** (NSApplication)

### nextEventMatchingMask:

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask*

Invokes NSApplication's **nextEventMatchingMask:untilDate:inMode:dequeue:** method, using *mask* as the first argument, with an unlimited expiration, a mode of NSEventTrackingRunLoopMode, and a dequeue flag of YES. See the method description in the NSApplication class specification for more information.

### nextEventMatchingMask:untilDate:inMode:dequeue:

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask*
**untilDate:**(NSDate *)*expirationDate*
**inMode:**(NSString *)*mode*
**dequeue:**(BOOL)*flag*

Forwards the message to the global NSApplication object, NSApp. See the method description in the NSApplication class specification for more information.

### orderBack:

– (void)**orderBack:**(id)*sender*

Moves the receiver to the back of its level in the screen list, without changing either the key window or the main window.

**See also:** – **orderFront:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**, – **level**

### orderFront:

– (void)**orderFront:**(id)*sender*

Moves the receiver to the front of its level in the screen list, without changing either the key window or the main window.

**See also:** – **orderBack:**, – **orderOut:**, – **orderWindow:relativeTo:**, – **makeKeyAndOrderFront:**, – **level**

### orderFrontRegardless

– (void)**orderFrontRegardless**

Moves the receiver to the front of its level, even if its application isn't active, but without changing either the key window or the main window. Normally an NSWindow can't be moved in front of the key window unless the NSWindow and the key window are in the same application. You should rarely need to invoke this method; it's designed to be used when applications are cooperating in such a way that an active application (with the key window) is using another application to display data.

**See also:** – **orderFront:**, – **level**

### orderOut:

– (void)**orderOut:**(id)*sender*

Takes the receiver out of the screen list. If the receiver is the key or main window, the NSWindow immediately below it is made key or main in its place.

**See also:** – **orderFront:**, – **orderBack:**, – **orderWindow:relativeTo:**

### orderWindow:relativeTo:

– (void)**orderWindow:**(NSWindowOrderingMode)*place* **relativeTo:**(int)*otherWindowNumber*

Repositions the receiver's window device in the Window Server's screen list. If *place* is NSWindowOut, the receiver is removed from the screen list and *otherWindowNumber* is ignored. If it's NSWindowAbove the receiver is ordered immediately above the window whose window number is *otherWindowNumber*. Similarly, if *place* is NSWindowBelow is placed immediately below the window represented by *otherWindowNumber*. If *otherWindowNumber* is 0, the receiver is placed above or below all other windows in its level.

**See also:** – **orderFront:**, – **orderBack:**, – **orderOut:**, – **makeKeyAndOrderFront:**, – **level**, – **windowNumber**

### performClose:

– (void)**performClose:**(id)*sender*

Simulates the user clicking the close button by momentarily highlighting the button and then closing the receiver. If the receiver's delegate or the receiver itself implements **windowShouldClose:**, then that message is sent with the NSWindow as the argument (only one such message is sent; if both the delegate and the NSWindow implement the method, only the delegate will receive the message). If the method returns NO, the NSWindow isn't closed.

If the receiver doesn't have a close button or can't be closed (for example, if the delegate replies NO to a **windowShouldClose:** message), then this method calls **NSBeep()**.

**See also:** – **close**, – **styleMask**, – **performClick:** (NSButton), – **performMiniaturize:**

### performMiniaturize:

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

Simulates the user clicking the miniaturize button by momentarily highlighting the button then miniaturizing the receiver. If the receiver doesn't have a miniaturize button or can't be miniaturized for some reason, then this method calls **NSBeep()**.

**See also:** – **close**, – **styleMask**, – **performClick:** (NSButton), – **performClose:**

### postEvent:atStart:

– (void)**postEvent:**(NSEvent *)*anEvent* **atStart:**(BOOL)*flag*

Forwards the message to the global NSApplication object, NSApp.

### print:

– (void)**print:**(id)*sender*

Runs the Print panel, and if the user chooses an option other than canceling, prints the receiver (its frame view and all subviews).

**See also:** – **fax:**

### registerForDraggedTypes:

– (void)**registerForDraggedTypes:**(NSArray *)*pboardTypes*

Registers *pboardTypes* as the pasteboard types that the receiver will accept as the destination of an image-dragging session.

**Note:** Registering an NSWindow for dragged types automatically makes it a candidate destination object for a dragging session. As such, it must properly implement some or all of the NSDraggingDestination protocol methods. As a convenience, NSWindow provides default implementations of these methods. See the NSDraggingDestination protocol specification for details.

**See also:** – **unregisterDraggedTypes**

### representedFilename

– (NSString *)**representedFilename**

Returns the name of the file that the receiver represents.

**See also:** – **setRepresentedFilename:**

### resetCursorRects

– (void)**resetCursorRects**

Invokes **discardCursorRects** to clear the receiver's cursor rectangles, then sends **resetCursorRects** to every NSView in the receiver's view hierarchy.

This method is typically invoked by the NSApplication object when it detects that the key window's cursor rectangles are invalid. In program code, it's more efficient to invoke **invalidateCursorRectsForView:**.

## resignKeyWindow

– (void)**resignKeyWindow**

Never invoke this method; it's invoked automatically when the NSWindow resigns key window status. This method sends **resignKeyWindow** to the receiver's first responder, sends **windowDidResignKey:** to the receiver's delegate, and posts an NSWindowDidResignKeyNotification to the default notification center.

**See also:** – **becomeKeyWindow**, – **resignMainWindow**

## resignMainWindow

– (void)**resignMainWindow**

Never invoke this method; it's invoked automatically when the NSWindow resigns main window status. This method sends **windowDidResignMain:** to the receiver's delegate and posts an NSWindowDidResignMainNotification to the default notification center.

**See also:** – **becomeMainWindow**, – **resignKeyWindow**

## resizeFlags

– (int)**resizeFlags**

Valid only while the receiver is being resized, this method returns the flags field of the event record for the mouse-down event that initiated the resizing session. The integer encodes, as a mask, which of the modifier keys was held down when the event occurred. The flags are listed in the NSEvent class's **modifierFlags** method description. You can use this method to constrain the direction or amount of resizing. Because of its limited validity, this method should only be invoked from within an implementation of the delegate method **windowWillResize:toSize:**.

## resizeIncrements

– (NSSize)**resizeIncrements**

Returns the receiver's resizing increments, which restrict the user's ability to resize it so that its width and height alter by integral multiples of *increments.width* and *increments.height* when the user resizes it. These amounts are whole number values, 1.0 or greater. You can set an NSWindow's size to any value programmatically.

**See also:** – **setResizeIncrements:**, – **setAspectRatio:**, – **setFrame:display:**

### restoreCachedImage

– (void)**restoreCachedImage**

Splices the receiver's cached image rectangles, if any, back into its raster image (and buffer if it has one), undoing the effect of any drawing performed within those areas since they were established using **cacheImageInRect:**. You must invoke **flushWindowIfNeeded** after this method to guarantee proper redisplay. An NSWindow automatically discards its cached image rectangles when it displays.

**See also:** – **discardCachedImage**, – **display**

### saveFrameUsingName:

– (void)**saveFrameUsingName:**(NSString *)*name*

Saves the receiver's frame rectangle in the user-defaults system. With the companion method **setFrameUsingName:**, you can save and reset an NSWindow's frame over various launchings of an application. The default is owned by the application and stored under the name "NSWindow Frame *name*". See the NSUserDefaults class specification for more information.

**See also:** – **setFrameUsingName:**, – **stringWithSavedFrame**

### screen

– (NSScreen *)**screen**

Returns the screen that the receiver is on. If the receiver is partly on one screen and partly on another, the screen where most of it lies is the one returned.

**See also:** – **bestScreen**

### selectKeyViewFollowingView:

– (void)**selectKeyViewFollowingView:**(NSView *)*aView*

Sends **nextValidKeyView:** to *aView*, and if that message returns an NSView, invokes **makeFirstResponder:** with the returned NSView.

**See also:** – **selectKeyViewPrecedingView:**

### selectKeyViewPrecedingView:

– (void)**selectKeyViewPrecedingView:**(NSView *)*aView*

Sends **previousValidKeyView:** to *aView*, and if that message returns an NSView, invokes **makeFirstResponder:** with the returned NSView.

**See also:** – **selectKeyViewFollowingView:**

### selectNextKeyView:

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

Searches for a candidate key view and, if it finds one, invokes **makeFirstResponder:** to establish it as the first responder. The candidate is one of (in order):

- The current first responder's next valid key view, as returned by NSView's **nextValidKeyView:** method.
- The object designated as the receiver's initial first responder (using **setInitialFirstResponder:**), if it returns YES to an **acceptsFirstResponder** message.
- Otherwise, the initial first responder's next valid key view, which may end up being **nil**.

**See also:** – **selectPreviousKeyView:**, – **selectKeyViewFollowingView:**

### selectPreviousKeyView:

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

Searches for a candidate key view and, if it finds one, invokes **makeFirstResponder:** to establish it as the first responder. The candidate is one of (in order):

- The current first responder's previous valid key view, as returned by NSView's **previousValidKeyView:** method.
- The object designated as the receiver's initial first responder (using **setInitialFirstResponder:**), if it returns YES to an **acceptsFirstResponder** message.
- Otherwise, the initial first responder's previous valid key view, which may end up being **nil**.

**See also:** – **selectNextKeyView:**, – **selectKeyViewPrecedingView:**

### sendEvent:

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

Dispatches mouse and keyboard events sent to the receiver by the NSApplication object. Never invoke this method directly.

### setAcceptsMouseMovedEvents:

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

Controls whether the receiver accepts mouse-moved events and distributes them to its responders. If *flag* is YES it does accept them, if *flag* is NO it doesn't. NSWindows by default don't accept mouse-moved events.

**See also:** – **acceptsMouseMovedEvents**

### setAspectRatio:

– (void)**setAspectRatio:**(NSSize)*ratio*

Sets the receiver's size aspect ratio to *ratio*, constraining the size of its frame rectangle to integral multiples of this size when the user resizes it. You can set an NSWindow's size to any ratio programmatically.

**See also:** – **setResizeIncrements:**, – **aspectRatio**, – **setFrame:display:**

### setAutodisplay:

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

Controls whether the receiver automatically displays its views that are marked as needing it. If *flag* is YES, views are automatically displayed as needed, typically on each pass through the event loop. If *flag* is NO, the receiver or its views must be explicitly displayed.

**See also:** – **isAutodisplay**, – **displayIfNeeded**, – **displayIfNeeded** (NSView)

### setBackgroundColor:

– (void)**setBackgroundColor:**(NSColor *)*aColor*

Sets the color of the receiver's background to *aColor*.

**See also:** – **backgroundColor**

### setBackingType:

– (void)**setBackingType:**(NSBackingStoreType)*backingType*

Sets the receiver's backing store type to *backingType*, which may be one of the following constants:

NSBackingStoreBuffered
NSBackingStoreRetained

This method can only be used to switch a buffered NSWindow to retained or vice versa; you can't change the backing type to or from nonretained after initializing an NSWindow (a PostScript error is generated if you attempt to do so).

**See also:** – **backingType**, – **initWithContentRect:...**

### setContentSize:

– (void)**setContentSize:**(NSSize)*aSize*

Sets the size of the receiver's content view to *aSize*, which is expressed in the receiver's base coordinate system. This in turn alters the size of the NSWindow itself. Note that the Window Server limits window sizes to 10,000; if necessary, be sure to limit *aSize* as needed relative to the frame rectangle.

**See also:** – **setFrame:display:**, + **contentRectForFrameRect:styleMask:**, + **frameRectForContentRect:styleMask**

### setContentView:

– (void)**setContentView:**(NSView *)*aView*

Makes *aView* the receiver's content view; the previous content view is removed from the receiver's view hierarchy and released. *aView* is resized to fit precisely within the content area of the NSWindow. You can modify the content view's coordinate system through its bounds rectangle, but can't alter its frame rectangle (that is, its size or location) directly.

This method causes the old content view to be released; if you plan to reuse it, be sure to retain it before sending this message and to release it as appropriate when adding it to another NSWindow or NSView.

**See also:** – **contentView**, – **setContentSize:**

### setDefaultButtonCell:

– (void)**setDefaultButtonCell:**(NSButtonCell *)*aButtonCell*

Makes *aButtonCell*'s key equivalent that for the Return (or Enter) key, so that when the user presses return that button performs as if clicked. See the method description for **defaultButtonCell** for more information.

**See also:** – **disableKeyEquivalentForDefaultButtonCell**, – **enableKeyEquivalentForDefaultButtonCell**

### setDelegate:

– (void)**setDelegate:**(id)*anObject*

Makes *anObject* the receiver's delegate, without retaining it. An NSWindow's delegate is inserted into the responder chain after the NSWindow itself, and is informed of various actions by the NSWindow through delegation messages.

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

### setDepthLimit:

– (void)**setDepthLimit:**(NSWindowDepth)*limit*

Sets the depth limit of the receiver to *limit*, which can be creating using the **NSBestDepth()** function. A value of 0 indicates the receiver's default depth limit; this can be useful for reverting an NSWindow to its initial depth.

**See also:** – **depthLimit**, + **defaultDepthLimit**, – **setDynamicDepthLimit:**

### setDocumentEdited:

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

Records whether the receiver's document has been edited but not saved. NSWindows are by default in "not edited" status.

You should invoke this method with an argument of YES every time the NSWindow's document changes in such a way that it needs to be saved and with an argument of NO every time it gets saved. Then, before closing the NSWindow you can use **isDocumentEdited** to determine whether to allow the user a chance to save the document.

### setDynamicDepthLimit:

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

Sets whether the receiver changes its depth to match the depth of the screen that it's on, or the deepest when it spans multiple screens. If *flag* is YES, the depth limit depends on which screen the receiver is on. If *flag* is NO, the receiver uses either its preset depth limit or the default depth limit. A different, and nondynamic, depth limit can be set with the **setDepthLimit:** method.

**See also:** – **hasDynamicDepthLimit**, + **defaultDepthLimit**

### setExcludedFromWindowsMenu:

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

Controls whether the receiver's title is omitted from the application's Windows menu. If *flag* is YES it's omitted; if *flag* is NO, it's listed when it or its miniwindow is on-screen. The default is NO.

**See also:** – **isExcludedFromWindowsMenu**

### setFrame:display:

– (void)**setFrame:**(NSRect)*frameRect* **display:**(BOOL)*flag*

Sets the origin and size of the receiver's frame rectangle according to *frameRect*, thereby setting its position and size on-screen, and invokes **display** if *flag* is YES. Note that the Window Server limits window position coordinates to ±16,000 and sizes to 10,000.

**See also:** – **frame**, – **setFrameFromString:**, – **setFrameOrigin:**, – **setFrameTopLeftPoint:**, – **setFrameUsingName:**

### setFrameAutosaveName:

– (BOOL)**setFrameAutosaveName:**(NSString *)*name*

Sets the name used to automatically save the receiver's frame rectangle in the defaults system to *name*. If *name* isn't the empty string (@""), the receiver's frame is saved as a user default (as described in **saveFrameUsingName:**) each time the frame changes. Returns YES if the name is set successfully, NO if it's being used as an autosave name by another NSWindow in the application (in which case the receiver's old name remains in effect).

**See also:** – **setFrameUsingName:**, + **removeFrameUsingName:**, – **stringWithSavedFrame**, – **setFrameFromString:**

### setFrameFromString:

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

Sets the receiver's frame rectangle from the string representation *aString*, a representation previously creating using **stringWithSavedFrame**. The frame is constrained according to the receiver's minimum and maximum size settings. This method causes a **windowWillResize:toSize:** message to be sent to the delegate.

**See also:** – **stringWithSavedFrame**

### setFrameOrigin:

– (void)**setFrameOrigin:**(NSPoint)*aPoint*

Positions the lower left corner of the receiver's frame rectangle at *aPoint* in screen coordinates. Note that the Window Server limits window position coordinates to ±16,000.

**See also:** – **setFrame:display:**, – **setFrameTopLeftPoint:**

### setFrameTopLeftPoint:

– (void)**setFrameTopLeftPoint:**(NSPoint)*aPoint*

Positions the top left corner of the receiver's frame rectangle at *aPoint* in screen coordinates. Note that the Window Server limits window position coordinates to ±16,000; if necessary, adjust *aPoint* relative to the window's lower left corner to account for this.

**See also:** – **cascadeTopLeftFromPoint:**, – **setFrame:display:**, – **setFrameOrigin:**

### setFrameUsingName:

– (BOOL)**setFrameUsingName:**(NSString *)*name*

Sets the receiver's frame rectangle by reading the rectangle data stored in *name* from the defaults system. The frame is constrained according to the receiver's minimum and maximum size settings. This method causes a **windowWillResize:toSize:** message to be sent to the delegate. Returns YES if *name* is read and the frame is set successfully; otherwise returns NO.

**See also:** – **setFrameAutosaveName:**, + **removeFrameUsingName:**, – **stringWithSavedFrame**, – **setFrameFromString:**

### setHidesOnDeactivate:

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

Determines whether the receiver is removed from the screen when the application is inactive. If *flag* is YES, the receiver is hidden (taken out of the screen list) when the application stops being the active application. If *flag* is NO, the receiver stays on-screen. The default for NSWindow is NO; the default for NSPanel is YES.

**See also:** – **hidesOnDeactivate:**

## setInitialFirstResponder:

– (void)**setInitialFirstResponder:**(NSView *)*aView*

Sets *aView* as the NSView that's made first responder (also called the key view) the first time the receiver is placed on-screen.

**See also:** – **initialFirstResponder**

## setLevel:

– (void)**setLevel:**(int)*newLevel*

Sets the receiver's window level to *newLevel*. Some useful predefined values are:

| Level | Comment |
|---|---|
| NSNormalWindowLevel | The default level for NSWindow objects |
| NSFloatingWindowLevel | Useful for floating palettes. |
| NSDockWindowLevel | Reserved for the application dock (not used on Microsoft Windows). |
| NSSubmenuWindowLevel | Reserved for submenus (not used on Microsoft Windows). |
| NSMainMenuWindowLevel | Reserved for the application's main menu (not used on Microsoft Windows). |

Each level in the list groups windows within it above those in all preceding groups. Floating windows, for example, appear above all normal level windows. When a window enters a new level it's ordered above all of its peers in that level.

**See also:** – **level**, – **orderWindow:relativeTo:**, – **orderFront:**, – **orderBack:**

## setMaxSize:

– (void)**setMaxSize:**(NSSize)*aSize*

Sets the maximum size to which the receiver's frame can be sized to *aSize*. The maximum size constraint is enforced for resizing by the user as well as for the **setFrame...** methods *other than* **setFrame:display:**. Note that the Window Server limits window sizes to 10,000.

**See also:** – **maxSize**, – **setMinSize:**, – **setAspectRatio:**, – **setResizeIncrements:**

### setMiniwindowImage:

– (void)**setMiniwindowImage:**(NSImage *)*anImage*

Sets the image displayed by the receiver's miniwindow to *anImage*.

**See also:** – **miniwindowImage**, – **isMiniaturized**

### setMiniwindowTitle:

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

Sets the title of the receiver's miniaturized counterpart to *aString* and redisplays it. A miniwindow's title normally reflects that of its full-size counterpart, abbreviated to fit if necessary. Although this method allows you to set the miniwindow's title explicitly, changing the full-size NSWindow's title (through **setTitle:** or **setTitleWithRepresentedFilename:**) automatically changes the miniwindow's title as well.

**See also:** – **miniwindowTitle**

### setMinSize:

– (void)**setMinSize:**(NSSize)*aSize*

Returns the minimum size to which the receiver's frame can be sized to *aSize*. The minimum size constraint is enforced for resizing by the user as well as for the **setFrame...** methods other than **setFrame:display:**.

**See also:** – **minSize**, – **setMaxSize:**, – **setAspectRatio:**, – **setResizeIncrements:**

### setOneShot:

– (void)**setOneShot:**(BOOL)*flag*

Sets whether the PostScript window device that the receiver manages should be freed when it's removed from the screen list (and another one created if it's returned to the screen). This can result in memory savings and performance improvement for NWindows that don't take long to display. It's particularly appropriate for NSWindows that the user might use once or twice but not display continually. The default is NO.

**See also:** – **isOneShot**

### setReleasedWhenClosed:

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

Determines the receiver's behavior when it receives a **close** message. If *flag* is NO, the receiver is merely hidden (taken out of the screen list). If *flag* is YES, the receiver is hidden and then released. The default for NSWindow is YES; the default for NSPanel is NO.

Another strategy for releasing an NSWindow is to have its delegate autorelease it on receiving a **windowShouldClose:** message.

**See also:** – **close**, – **isReleasedWhenClosed**

### setRepresentedFilename:

– (void)**setRepresentedFilename:**(NSString *)*path*

Sets the name of the file that the receiver represents to *path*.

**See also:** – **representedFilename:**, – **setTitleWithRepresentedFilename:**

### setResizeIncrements:

– (void)**setResizeIncrements:**(NSSize)*increments*

Restricts the user's ability to resize the window so that its width and height alter by multiples of *increments.width* and *increments.height* as the user resizes it. These amounts should be whole numbers, 1.0 or greater. You can set an NSWindow's size to any ratio programmatically.

**See also:** – **resizeIncrements**, – **setAspectRatio:**, – **setFrame:display:**

### setTitle:

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

Sets the string that appears in the receiver's title bar (if it has one) to *aString* and displays the title. Also sets the title of the receiver's miniwindow.

**See also:** – **title**, – **setTitleWithRepresentedFilename:**, – **setMiniwindowTitle:**

### setTitleWithRepresentedFilename:

– (void)**setTitleWithRepresentedFilename:**(NSString *)*path*

Sets *path* as the receiver's title, formatting it as a file system path, and records *path* as the receiver's associated filename using **setRepresentedFilename:**. The title format varies with the platform. On

Mach-based systems, the file name is displayed first, followed by an em dash and the path for the directory containing the file. The em dash is offset by two spaces on either side. For example:

MyFile — /Net/server/group/home

This method also sets the title of the receiver's miniwindow.

**See also:** – **title**, – **setTitle:**, – **setMiniwindowTitle:**, – **setRepresentedFilename:**

### setViewsNeedDisplay:

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

Records the receiver's display status. If *flag* is YES, the receiver records itself as needing display. If *flag* is NO the receiver records its NSViews as not needing display. You should rarely need to invoke this method; NSView's **setNeedsDisplay:** and similar methods invoke it automatically.

**See also:** – **viewsNeedDisplay**

### stringWithSavedFrame

– (NSString *)**stringWithSavedFrame**

Returns a string that represents the receiver's frame rectangle in a format that can be used with a later **setFrameUsingName:** message.

### styleMask

– (unsigned int)**styleMask**

Returns the receiver's style mask, indicating what kinds of control items it displays. See the description for the style mask in the **initWithContentRect:styleMask:backing:defer:** method description. An NSWindow's style is set when the object is initialized. Once set, it can't be changed.

### title

– (NSString *)**title**

Returns the string that appears in the title bar of the receiver.

**See also:** – **setTitle:**, – **setTitleWithRepresentedFilename:**

### tryToPerform:with:

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

Overrides NSResponder's implementation to add the receiver's delegate to the series of objects requested to perform *anAction*. Returns YES if a receiver for *anAction* is found; otherwise returns NO.

**See also:** – **tryToPerform:with:** (NSResponder)

### unregisterDraggedTypes

– (void)**unregisterDraggedTypes**

Unregisters the receiver as a possible destination for dragging operations.

**See also:** – **registerForDraggedTypes:**

### update

– (void)**update**

The default implementation of this method does nothing more than post an NSWindowDidUpdateNotification to the default notification center. A subclass can override this method to perform specialized operations, but should send an **update** message to **super** just before returning. For example, the NSMenu class implements this method to disable and enable menu commands.

An NSWindow is automatically sent an **update** message on every pass through the event loop and before it's ordered on-screen. You can manually cause an update message to be sent to all visible NSWindows through NSApplication's **updateWindows** method.

**See also:** – **setWindowsNeedUpdate:** (NSApplication), – **updateWindows** (NSApplication)

### useOptimizedDrawing:

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

Informs the receiver whether to optimize focusing and drawing when displaying its NSViews. The optimizations may prevent sibling subviews from being displayed in the correct order—which matters only if the subviews overlap. You should always set *flag* to YES if there are no overlapping subviews within the NSWindow. The default is NO.

### validRequestorForSendType:returnType:

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

Searches for an object that responds to a Services request by providing input of *sendType* and accepting output of *returnType*. Returns that object, or **nil** if none is found.

Messages to perform this method are initiated by the Services menu. It's part of the mechanism that passes **validRequestorForSendType:andReturnType:** messages up the responder chain. See the Services documentation in *Programming Topics* for more information.

This method works by forwarding the message to the receiver's delegate if it responds (and provided it isn't an NSResponder with its own next responder). If the delegate doesn't respond to the message or returns **nil** when sent it, this method forwards the message to the NSApplication object. If the NSApplication object returns **nil**, this method also returns **nil**. Otherwise this method returns the object returned by the delegate or NSApplication object.

**See also:** – **validRequestorForSendType:returnType:** (NSResponder and NSApplication)

### viewsNeedDisplay

– (BOOL)**viewsNeedDisplay**

Returns YES if any of the receiver's NSView's need to be displayed, NO otherwise.

**See also:** – **setViewsNeedDisplay:**

### windowHandle

– (void *)**windowHandle**

Returns a Microsoft Windows HWND handle as a pointer to **void**. This value can be cast directly to HWND. This method exists only on Microsoft Windows; don't attempt to invoke it on Mach.

### windowNumber

– (int)**windowNumber**

Returns the window number of the receiver's PostScript window device. Each window device in an application is given a unique window number—note that this isn't the same as the global window number assigned by the Window Server. This number can be used to identify the window device with the **orderWindow:relativeTo:** method and in the Application Kit functions **NSWindowList()** and **NSConvertWindowNumberToGlobal()**.

If the receiver doesn't have a window device, the value returned will be equal to or less than 0.

**See also:** – **initWithContentRect:styleMask:backing:defer:**, – **setOneShot:**

### worksWhenModal

– (BOOL)**worksWhenModal**

Returns YES if the receiver is able to receive keyboard and mouse events even when some other window is being run modally, NO otherwise. NSWindow's implementation of this method returns NO. Only subclasses of NSPanel should override this default.

**See also:** – **setWorksWhenModal:** (NSPanel)

## Methods Implemented By the Delegate

### windowDidBecomeKey:

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

Sent by the default notification center immediately after an NSWindow has become key. *aNotification* is always NSWindowDidBecomeKeyNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification.*

### windowDidBecomeMain:

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

Sent by the default notification center immediately after an NSWindow has become main. *aNotification* is always NSWindowDidBecomeMainNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification.*

### windowDidChangeScreen:

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

Sent by the default notification center immediately after an NSWindow has changed screen. *aNotification* is always NSWindowDidChangeScreenNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification.*

### windowDidDeminiaturize:

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

Sent by the default notification center immediately after an NSWindow has been deminiaturized. *aNotification* is always NSWindowDidDeminiaturizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowDidExpose:

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

Sent by the default notification center immediately after an NSWindow has been exposed. *aNotification* is always NSWindowDidExposeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowDidMiniaturize:

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

Sent by the default notification center immediately after an NSWindow has been miniaturized. *aNotification* is always NSWindowDidMiniaturizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowDidMove:

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

Sent by the default notification center immediately after an NSWindow has been moved. *aNotification* is always NSWindowDidMoveNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification.*

### windowDidResignKey:

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

Sent by the default notification center immediately after an NSWindow has resigned its status as key window. *aNotification* is always NSWindowDidResignKeyNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowDidResignMain:

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

Sent by the default notification center immediately after an NSWindow has resigned its status as main window. *aNotification* is always NSWindowDidResignMainNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowDidResize:

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

Sent by the default notification center immediately after an NSWindow has been resized. *aNotification* is always NSWindowDidResizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowDidUpdate:

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

Sent by the default notification center immediately after an NSWindow receives an **update** message. *aNotification* is always NSWindowDidUpdateNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowShouldClose:

– (BOOL)**windowShouldClose:**(id)*sender*

Invoked just before the user (or the **performClose:** method) closes *sender*. The delegate can return NO to prevent *sender* from closing.

### windowWillClose:

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

Sent by the default notification center immediately before an NSWindow closes. *aNotification* is always NSWindowWillCloseNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowWillMiniaturize:

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

Sent by the default notification center immediately before an NSWindow is miniaturized. *aNotification* is always NSWindowWillMiniaturizeNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification.*

### windowWillMove:

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

Sent by the default notification center immediately before an NSWindow is moved. *aNotification* is always NSWindowWillMoveNotification. You can retrieve the NSWindow object in question by sending **object** to *aNotification*.

### windowWillResize:toSize:

– (NSSize)**windowWillResize:**(NSWindow *)*sender* **toSize:**(NSSize)*proposedFrameSize*

Invoked *sender* is being resized (whether by the user or through one of the **setFrame...** methods other than **setFrame:display:**). *proposedFrameSize* contains the size (in screen coordinates) that the *sender* will be resized to. To reset the size, simply return *sender* directly from this method. The NSWindow's minimum and maximum size constraints have already been applied when this method is invoked.

While the user is resizing an NSWindow, the delegate is sent a series of **windowWillResize:toSize:** messages as the NSWindow's outline is dragged. The NSWindow's outline is displayed at the constrained size as set by this method.

### windowWillReturnFieldEditor:toObject:

– (id)**windowWillReturnFieldEditor:**(NSWindow *)*sender* **toObject:**(id)*anObject*

Invoked when *sender*'s field editor is requested by *anObject*. If the delegate's implementation of this method returns an object other than **nil**, the NSWindow substitutes it for the field editor and returns it to *anObject*.

**See also:** – **fieldEditor:forObject**

## Notifications

### NSWindowDidBecomeKeyNotification

Posted whenever the NSWindow becomes the key window. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has become key |
| Userinfo | None |

### NSWindowDidBecomeMainNotification

Posted whenever the NSWindow becomes the main window. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has become main |
| Userinfo | None |

### NSWindowDidChangeScreenNotification

Posted whenever a portion of the NSWindow's frame moves onto or off of a screen. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has changed screen |
| Userinfo | None |

### NSWindowDidDeminiaturizeNotification

Posted whenever the NSWindow is deminiaturized. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has deminiaturized |
| Userinfo | None |

### NSWindowDidExposeNotification

Posted whenever a portion of a nonretained NSWindow is exposed, whether by being ordered above other windows or by other widows being removed from above it. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has been exposed |
| Userinfo | |
| **Key** | **Value** |
| NSExposedRect | The rectangle that has been exposed |

### NSWindowDidMiniaturizeNotification

Posted whenever the NSWindow is miniaturized. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has miniaturized |
| Userinfo | None |

### NSWindowDidMoveNotification

Posted whenever the NSWindow is moved. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has moved |
| Userinfo | None |

### NSWindowDidResignKeyNotification

Posted whenever the NSWindow resigns its status as key window. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has resigned its key window status |
| Userinfo | None |

### NSWindowDidResignMainNotification

Posted whenever the NSWindow resigns its status as main window. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that has resigned its main window status |
| Userinfo | None |

### NSWindowDidResizeNotification

Posted whenever the NSWindow's size changes. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow whose size has changed |
| Userinfo | None |

### NSWindowDidUpdateNotification

Posted whenever the NSWindow receives an **update** message. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that received the **update** message |
| Userinfo | None |

### NSWindowWillCloseNotification

Posted whenever the NSWindow is about to close. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that’s closing |
| Userinfo | None |

### NSWindowWillMiniaturizeNotification

Posted whenever the NSWindow is about to miniaturize. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that’s miniaturizing |
| Userinfo | None |

### NSWindowWillMoveNotification

Posted whenever the NSWindow is about to move. The notification contains:

| | |
|---|---|
| **Notification Object** | The NSWindow that’s moving |
| Userinfo | None |