NSApplication

Inherits From: NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSApplication.h
AppKit/NSColorPanel.h
AppKit/NSDataLinkPanel.h
AppKit/NSHelpManager.h
AppKit/NSPageLayout.h

Class at a Glance

Purpose

An NSApplication object manages an application's main event loop in addition to resources used by all of that application's objects.

Principal Attributes

- Delegate
- Key window
- DPS context
- List of windows
- Main window

Creation

Project Builder

+ sharedApplication	Creates the shared application instance (global variable NSApp).

Commonly Used Methods

– keyWindow	Returns an NSWindow representing the key window.
– mainWindow	Returns an NSWindow representing the main window.
– registerServicesMenuSendTypes:returnTypes:	Specifies which services are valid for this application.
– runModalForWindow:	Runs a modal event loop for the specified NSWindow.

Class Description

The NSApplication class provides the central framework for your application's execution. Every application must have exactly one instance of NSApplication (or a subclass of NSApplication). Your program's **main()** function should create this instance by invoking the **sharedApplication** class method. After creating the NSApplication object, the **main()** function should load your application's main nib file and then start the event loop by sending the NSApplication object a **run** message. If you create an Application project in Project Builder, this **main()** function is created for you. The **main()** function that Project Builder creates begins by calling a function named **NSApplicationMain()**, which is functionally similar to the following:

```
void NSApplicationMain(int argc, char *argv[]) {
    [NSApplication sharedApplication];
    [NSBundle loadNibNamed:@"myMain" owner:app];
    [NSApp run];
}
```

The **sharedApplication** class method initializes the PostScript environment and connects your program to the Window Server and the Display PostScript server. The NSApplication object maintains a list of all the NSWindows that the application uses, so it can retrieve any of the application's NSViews. **sharedApplication** also initializes the global variable NSApp, which you use to retrieve the NSApplication instance. **sharedApplication** only performs the initialization once; if you invoke it more than once, it simply returns the NSApplication object that it created previously.

NSApplication's main purpose is to receive events from the Window Server and distribute them to the proper NSResponders. NSApp translates an event into an NSEvent object, then forwards the NSEvent to the affected NSWindow object. All keyboard and mouse events go directly to the NSWindow associated with the event. The only exception to this rule is if the Command key is pressed when a key-down event occurs; in this case, every NSWindow has an opportunity to respond to the event. When an NSWindow receives an NSEvent from NSApp, it distributes it to the objects in its view hierarchy.

The NSApplication class sets up autorelease pools (instances of the NSAutoreleasePool class) during initialization and inside the event loop—specifically, within its **init** (or **sharedApplication**) and **run** methods. Similarly, the methods that the Application Kit adds to NSBundle employ autorelease pools during the loading of nib files. These autorelease pools aren't accessible outside the scope of the respective NSApplication and NSBundle methods. Typically, an application creates objects either while the event loop is running or by loading objects from nib files, so this usually isn't a problem. However, if you do need to use OpenStep classes within the **main()** function itself (other than to load nib files or to instantiate NSApplication), you should create an autorelease pool before using the classes and then release the pool when you're done. For more information, see the NSAutoreleasePool class specification in the *Foundation Framework Reference*.

Subclassing NSApplication

Rarely do you need to create a custom NSApplication subclass. In general, a better design is to separate the code that embodies your program's functionality into a number of custom objects. Usually, those custom

objects are subclasses of NSObject. Methods defined in your custom objects can be invoked from a small dispatcher object without being closely tied to NSApp. The only reason to subclass NSApplication is if you need to provide your own special response to messages that are routinely sent to NSApp. (Even then, NSApp's delegate is often given a chance to respond to such messages, so it's more appropriate to implement the delegate methods.) To use a custom subclass of NSApplication, simply send **sharedApplication** to your custom class rather than directly to NSApplication. If you create your application in Project Builder, set the application class on the Project Attributes inspector, and Project Builder will update the **main()** function accordingly. As mentioned previously, NSApp uses autorelease pools in its **init** and **run** methods; if you override these methods, you'll need to create your own autorelease pools.

The Delegate and Notifications

You can assign a delegate to NSApp. The delegate responds to certain messages on behalf of NSApp. Some of these messages, such as **application:openFile:**, ask the delegate to open a file. Another message, **applicationShouldTerminate:**, lets the delegate determine whether the application should be allowed to quit. The NSApplication class sends these messages directly to its delegate.

NSApp also posts notifications to the application's default notification center. Any object may register to receive one or more of the notifications posted by NSApp by sending the message **addObserver:selector:name:object:** to the default notification center (an instance of the NSNotificationCenter class). NSApp's delegate is automatically registered to receive these notifications if it implements certain delegate methods. For example, NSApp posts notifications when it is about to be done launching the application and when it is done launching the application (NSApplicationWillFinishLaunchingNotification and NSApplicationDidFinishLaunchingNotification). The delegate has an opportunity to respond to these notifications by implementing the methods **applicationWillFinishLaunching:** and **applicationDidFinishLaunching:**. If the delegate wants to be informed of both events, it implements both methods. If it only needs to know when the application is finished launching, it implements only **applicationDidFinishLaunching:**. For more information on notifications, see the NSNotificationCenter class specification in the *Foundation Framework Reference*.

Method Types

Creating and initializing an NSApplication	+ sharedApplication
	– finishLaunching
Changing the active application	– activateIgnoringOtherApps:
	– isActive
	– deactivate

Running the event loop	– run – isRunning – stop: – runModalForWindow: – stopModal – stopModalWithCode: – abortModal – beginModalSessionForWindow: – runModalSession: – endModalSession: – sendEvent:
Getting, removing, and posting events	– currentEvent – nextEventMatchingMask:untilDate:inMode:dequeue: – discardEventsMatchingMask:beforeEvent: – postEvent:atStart:
Managing windows	– keyWindow – mainWindow – windowWithWindowNumber: – windows – makeWindowsPerform:inOrder: – setWindowsNeedUpdate: – updateWindows – miniaturizeAll: – preventWindowOrdering
Hiding all windows	– hide: – isHidden – unhide: – unhideWithoutActivation
Setting the application’s icon	– setApplicationIconImage: – applicationIconImage
Getting the main menu	– setMainMenu: – mainMenu
Managing the Window menu	– setWindowsMenu: – windowsMenu – arrangeInFront: – addWindowsItem:title:filename: – changeWindowsItem:title:filename: – removeWindowsItem: – updateWindowsItem:

Managing the Services menu	– setServicesMenu: – servicesMenu – registerServicesMenuSendTypes:returnTypes: – validRequestorForSendType:returnType: – setServicesProvider: – servicesProvider:
Showing standard panels	– orderFrontColorPanel: – orderFrontDataLinkPanel: – runPageLayout:
Displaying help	– showHelp: – activateContextHelpMode:
Sending action messages	– sendAction:to:from: – tryToPerform:with: – targetForAction:
Getting the Display PostScript context	– context
Reporting an exception	– reportException:
Terminating the application	– terminate:
Assigning a delegate	– setDelegate: – delegate
Microsoft Windows® specific methods	– applicationHandle – windowWithWindowHandle: + setApplicationHandle:previousHandle:commandLine:show: + useRunningCopyOfApplication

Class Methods

setApplicationHandle:previousHandle:commandLine:show:

+ (void)**setApplicationHandle:**(void *)*hInstance*
previousHandle:(void *)*prevInstance*
commandLine:(NSString *)*cmdLine*
show:(int)*cmdShow*

On Microsoft Windows platforms, informs the NSApplication class of the values for the arguments passed to the **WinMain()** function. This message should be sent once, as the first line of the **WinMain()** function. If you create your application using Project Builder, this is done for you. You only need to invoke this method if you implement your own **WinMain()** function. Don’t override this method in NSApplication subclasses.

This method is not implemented on the Mach platform.

See also: – **applicationHandle**

sharedApplication

+ (NSApplication *)**sharedApplication**

Returns the NSApplication instance (the global NSApp), creating it if it doesn't exist yet. This method also makes a connection to the Window Server and completes other initialization. Your program should invoke this method as one of the first statements in **main()**; this is done for you if you create your application with Project Builder. To retrieve the NSApplication instance after it has been created, you use the global variable NSApp or invoke this method.

See also: – **run**, – **terminate:**

useRunningCopyOfApplication

+ (void)**useRunningCopyOfApplication**

On Microsoft Windows platforms, attempts to find an already running copy of the application at startup. This method is invoked in the **WinMain()** function. If the command used to start the application contains the option **-NSUseRunningCopy YES** and the application is already running, this method causes that version of the application to be activated rather than start up a new copy.

The method returns if the **-NSUseRunningCopy YES** option was not specified, if there was no previously running copy, or if the running copy was unable to be used (for any reason). If a running copy is successfully found and used, this method exits with a code of 0.

You never need to invoke this method directly. If you need to prevent the system from using an already running copy of the application, write your own **WinMain()** function, removing this method invocation. NSApplication subclasses should not override this method.

This method is not defined for the Mach platform.

Instance Methods

abortModal

– (void)**abortModal**

Aborts the event loop started by **runModalForWindow:** by raising an NSAbortModalException, which is caught by **runModalForWindow:**. Because this method raises an exception, it never returns; **runModalForWindow:**, when stopped with this method, returns NSRunAbortedResponse. **abortModal**

is typically sent by objects registered with the default NSRunLoop; for example, by objects that have registered a method to be repeatedly invoked by the NSRunLoop through the use of an NSTimer object.

This method can also abort a modal session created by **beginModalSessionForWindow:**, provided the loop that runs the modal session (by invoking **runModalSession:**) catches NSAbortModalException.

See also: – **endModalSession:**, – **stopModal**, – **stopModalWithCode:**

activateContextHelpMode:

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

Places the application in context-sensitive help mode. In this mode, the cursor becomes a question mark, and help appears for any user interface item that the user clicks. This method is typically invoked on Microsoft Windows platforms when the user selects the What's This menu item. (An application also enters context-sensitive help mode on Microsoft Windows platforms when the user presses Shift-F1.)

On Mach platforms, most applications don't use this method. Instead, applications enter context-sensitive mode when the user presses the Help key. On either platform, applications exit context-sensitive help mode upon the first event after a help window is displayed.

See also: – **showHelp:**

activateIgnoringOtherApps:

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

Makes the receiver the active application. If *flag* is NO, the application is activated only if no other application is currently active. If *flag* is YES, the application activates regardless.

On Mach platforms, *flag* is normally set to NO. When the Workspace Manager launches an application, using a value of NO for *flag* allows the application to become active if the user waits for it to launch, but the application remains unobtrusive if the user activates another application. Regardless of the setting of *flag*, there may be a time lag before the application activates; you should not assume that the application will be active immediately after sending this message.

On Microsoft Windows platforms, *flag* is normally set to YES. Setting *flag* to NO has no effect.

You rarely need to invoke this method. Under most circumstances, the Application Kit takes care of proper activation. However, you might find this method useful if you implement your own methods for interapplication communication.

You don't need to send this message to make one of the application's NSWindows key. When you send a **makeKeyWindow** message to an NSWindow, you simply ensure that the NSWindow will be the key window when the application is active.

See also: – **deactivate**, – **isActive**

addWindowsItem:title:filename:

– (void)**addWindowsItem:**(NSWindow *)*aWindow*
title:(NSString *)*aString*
filename:(BOOL)*isFilename*

Adds an item to the Window menu for *aWindow*. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted path name with the name of the file preceding the path (the way NSWindow's **setTitleWithRepresentedFilename:** method shows a title). If an item for *aWindow* already exists in the Window menu, this method has no effect. You rarely invoke this method because an item is placed in the Window menu for you whenever an NSWindow's title is set.

See also: – **changeWindowsItem:title:filename:**, – **setTitle:** (NSWindow),
– **setTitleWithRepresentedFilename:** (NSWindow)

applicationHandle

- (void *)**applicationHandle**

On Microsoft Windows platforms, returns the application's Win32 instance handle, which is a required parameter for some Win32 function calls. This method is not defined for the Mach platform.

See also: + **setApplicationHandle:previousHandle:commandLine:show:**

applicationIconImage

– (NSImage *)**applicationIconImage**

Returns the NSImage used for the application's icon, which represents the application in the Workspace Manager on Mach platforms or in the Program Manager on Microsoft Windows platforms.

See also: – **setApplicationIconImage:**

arrangeInFront:

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

Arranges all of the windows listed in the Window menu in front of all other windows. Windows associated with the application but not listed in the Window menu are not ordered to the front.

See also: – **addWindowItem:title:filename:**, – **removeWindowsItem:**,
– **makeKeyAndOrderFront:** (NSWindow)

beginModalSessionForWindow:

– (NSModalSession)**beginModalSessionForWindow:**(NSWindow *)*aWindow*

Sets up a modal session with the NSWindow *aWindow* and returns an NSModalSession structure representing the session. In a modal session, the application receives mouse events only if they occur in *aWindow*. The NSWindow is made key and ordered to the front.

beginModalSessionForWindow: only sets up the modal session. To actually run the session, use **runModalSession:**. **beginModalSessionForWindow:** should be balanced by **endModalSession:**. If an exception is raised, **beginModalSessionForWindow:** arranges for proper cleanup. Do *not* use NS_DURING constructs to send an **endModalSession:** message in the event of an exception.

A loop using these methods is similar to a modal event loop run with **runModalForWindow:**, except that the application can continue processing between method invocations.

changeWindowsItem:title:filename:

– (void)**changeWindowsItem:**(NSWindow *)*aWindow* **title:**(NSString *)*aString*
filename:(BOOL)*isFilename*

Changes the item for *aWindow* in the Window menu to *aString*. If *aWindow* doesn't have an item in the Window menu, this method adds the item. If *isFilename* is NO, *aString* appears literally in the menu. If *isFilename* is YES, *aString* is assumed to be a converted path name with the file's name preceding the path (the way NSWindow's **setTitleWithRepresentedFilename:** places a title).

See also: – **addWindowsItem:title:filename:**, – **removeWindowsItem:**, – **setTitle:** (NSWindow), – **setTitleWithRepresentedFilename:** (NSWindow)

context

– (NSDPSContext *)**context**

Returns the receiver's Display PostScript context.

currentEvent

– (NSEvent *)**currentEvent**

Returns the current event, the last event the receiver retrieved from the event queue. NSApp receives events and forwards the current event to the affected NSWindow object, which then distributes it to the objects in its view hierarchy.

See also: – **discardEventsMatchingMask:beforeEvent:**, – **postEvent:atStart:**, – **sendEvent:**

deactivate

– (void)**deactivate**

Deactivates the application. Normally, you shouldn't invoke this method; the Application Kit is responsible for proper deactivation.

See also: – **activateIgnoringOtherApps:**

delegate

– (id)**delegate**

Returns the receiver's delegate.

See also: – **setDelegate:**

discardEventsMatchingMask:beforeEvent:

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

Removes from the event queue all events matching those specified in *mask* that were generated before *lastEvent*. Typically, you send this message to an NSWindow rather than to NSApp.

mask can contain these constants:

NSLeftMouseDownMask	The left mouse button was pressed.
NSLeftMouseUpMask	The left mouse button was released.
NSRightMouseDownMask	The right mouse button was pressed.
NSRightMouseUpMask	The right mouse button was released.
NSMouseMovedMask	The user moved the mouse.
NSLeftMouseDraggedMask	The user moved the mouse while the left button was pressed.
NSRightMouseDraggedMask	The user moved the mouse while the right button was pressed.
NSMouseEnteredMask	The mouse entered a tracking rectangle.
NSMouseExitedMask	The mouse exited a tracking rectangle.
NSKeyDownMask	A key on the keyboard was pressed.
NSKeyUpMask	A key on the keyboard was released.
NSFlagsChangedMask	A Shift, Command, Alternate, or Escape key was pressed or released.
NSPeriodicMask	A periodic event occurred.
NSCursorUpdateMask	Cursor update.
NSAnyEventMask	Any event.

Use this method to ignore certain events that occurred after a particular event. For example, suppose your application has a tracking loop that you exit when the user releases the mouse button, and you want to discard all of the events that occurred during that loop. You use NSAnyEvent as the *mask* argument and pass the mouse up event as the *lastEvent* argument. Passing the mouse-up event as *lastEvent* ensures that any events that might have occurred after the mouse-up event (that is, that appear in the queue after the mouse-up event) don't get discarded.

See also: – **nextEventMatchingMask:untilDate:inMode:dequeue:**

encodeWithCoder:

@protocol NSCoding
– (void)**encodeWithCoder:**(NSCoder *)*aCoder*

Raises an NSInvalidArgumentException. You cannot encode an NSApplication instance.

See also: – **initWithCoder:**

endModalSession:

– (void)**endModalSession:**(NSModalSession)*session*

Finishes a modal session. The argument *session* should be the return value from a previous invocation of **beginModalSessionForWindow:**.

See also: – **runModalSession:**

finishLaunching

– (void)**finishLaunching**

Activates the application, opens any files specified by the "NSOpen" user default, and unhighlights the application's icon. The **run** method invokes this method before it starts the event loop. When this method begins, it posts an NSApplicationWillFinishLaunchingNotification to the default notification center. When it successfully completes, it posts an NSApplicationDidFinishLaunchingNotification. If you override **finishLaunching**, the subclass method should invoke the superclass method.

See also: – **applicationWillFinishLaunching:** (delegate method),
– **applicationDidFinishLaunching:** (delegate method)

hide:

– (void)**hide:**(id)*sender*

Hides all the application's windows. This method is usually invoked when the user chooses Hide in the application's main menu. When this method begins, it posts an NSApplicationWillHideNotification to the default notification center. When it completes successfully, it posts an NSApplicationDidHideNotification.

See also: – **applicationDidHide:** (delegate method), – **applicationWillHide:** (delegate method),
– **miniaturizeAll:**, – **unhide:**, – **unhideWithoutActivation**

initWithCoder:

@protocol NSCoding
– (id)**initWithCoder:**(NSCoder *)*aDecoder*

Raises an NSInvalidArgumentException. You cannot encode an NSApplication instance.

See also: – **encodeWithCoder:**

isActive

– (BOOL)**isActive**

Returns YES if this is the active application, NO otherwise.

See also: – **activateIgnoringOtherApps:**, – **deactivate**

isHidden

– (BOOL)**isHidden**

Returns YES if the application is hidden, NO otherwise.

See also: – **hide:**, – **unhide:**, – **unhideWithoutActivation**

isRunning

– (BOOL)**isRunning**

Returns YES if the main event loop is running, NO otherwise. NO means the **stop:** method was invoked.

See also: – **run**, – **terminate:**

keyWindow

– (NSWindow *)**keyWindow**

Returns the key window, the NSWindow that receives keyboard events. If there is no key window or if the key window belongs to another application, this method returns **nil**.

See also: – **mainWindow**, – **isKeyWindow** (NSWindow)

mainMenu

– (NSMenu *)**mainMenu**

Returns the application's main menu.

See also: – **setMainMenu:**

mainWindow

– (NSWindow *)**mainWindow**

Returns the main window. If there is no main window, if the main window belongs to another application, or if the application is hidden, this method returns **nil**.

See also: – **keyWindow**, – **isMainWindow** (NSWindow)

makeWindowsPerform:inOrder:

– (NSWindow *)**makeWindowsPerform:**(SEL)*aSelector* **inOrder:**(BOOL)*flag*

Sends the *aSelector* message to each NSWindow in the application in turn until one of them returns a value other than **nil**. Returns that NSWindow, or **nil** if all of the NSWindows returned **nil** for *aSelector*.

If *flag* is YES, the NSWindows receive the *aSelector* message in the front-to-back order in which they appear in the Window Server's window list. If *flag* is NO, NSWindows receive the message in the order they appear in NSApp's window list. This order is unspecified.

The method designated by *aSelector* can't take any arguments.

See also: – **sendAction:to:from:**, – **tryToPerform:with:**, – **windows**

miniaturizeAll:

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

Miniaturizes all the receiver's windows.

See also: – **hide:**

nextEventMatchingMask:untilDate:inMode:dequeue:

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

Returns the next event matching *mask*, or **nil** if no such event is found before the *expiration* date. If *flag* is YES, the event is removed from the queue. See the method description for **discardEventsMatchingMask:beforeEvent:** for a list of the possible values for *mask*.

The *mode* argument names an NSRunLoop mode that determines what other ports are listened to and what timers may fire while NSApp is waiting for the event. The possible modes available in the Application Kit are:

NSDefaultRunLoopMode Main event loop.

NSEventTrackingRunLoopModeModal event loops.

NSModalPanelRunLoopModeLoops that operate while a modal panel is up.

NSConnectionReplyMode Loops that operate while NSConnection is waiting for reply.

Events that are skipped are left in the queue.

You can use this method to short circuit normal event dispatching and get your own events. For example, you may want to do this in response to a mouse-down event in order to track the mouse while it's down. In this case, you would set *mask* to accept mouse-dragged or mouse-up events and use the NSEventTrackingRunLoopMode.

See also: – **postEvent:atStart:**, – **run**, – **runModalForWindow:**

orderFrontColorPanel:

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

Brings up the color panel, an instance of NSColorPanel. If the NSColorPanel does not exist yet, it creates one. This method is typically invoked when the user chooses Colors from a menu.

orderFrontDataLinkPanel:

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

Brings up the data link panel, an instance of NSDataLinkPanel. If the NSDataLinkPanel does not exist yet, it creates one. This method is typically invoked when the user chooses an appropriate command from the application's menu. For example, the Edit application invokes this method when the user chooses Link Inspector from the Link menu.

postEvent:atStart:

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

Adds *anEvent* to the application's event queue. If *flag* is YES, the event is added to the front of the queue, otherwise the event is added to the back of the queue.

See also: – **currentEvent**, – **sendEvent:**

preventWindowOrdering

– (void)**preventWindowOrdering**

Suppresses the usual window ordering in handling the most recent mouse-down event. This method is only useful for mouse-down events when you want to prevent the window that receives the event from being ordered to the front.

registerServicesMenuSendTypes:returnTypes:

– (void)**registerServicesMenuSendTypes:**(NSArray *)*sendTypes*
returnTypes:(NSArray *)*returnTypes*

Registers the pasteboard types that the application can send and receive in response to service requests. If the application has a Services menu, a menu item is added for each service provider that can accept one of the specified *sendTypes* or return one of the specified *returnTypes*. You should typically invoke this method at application start-up time or when an object that can use services is created. You can invoke it more than once; its purpose is to ensure that there is a menu item for every service that the application may use. The event-handling mechanism will dynamically enable the individual items to indicate which services are currently appropriate. All of the NSResponders in your application (typically NSViews) should register every possible type that they can send and receive by sending this message to NSApp.

See also: – **validRequestorForSendType:returnType:**,
– **readSelectionFromPasteboard:** (NSServicesRequests protocol),
– **writeSelectionToPasteboard:types:** (NSServicesRequests protocol)

removeWindowsItem:

– (void)**removeWindowsItem:**(NSWindow *)*aWindow*

Removes the Window menu item for *aWindow*. This method doesn't prevent the item from being automatically added again. Use NSWindow's **setExcludedFromWindowsMenu:** method if you want the item to remain excluded from the Window menu.

See also: – **addWindowsItem:title:filename:**, – **changeWindowsItem:title:filename:**

reportException:

– (void)**reportException:**(NSException *)*anException*

Logs *anException* by calling **NSLog()**. This method does not raise the exception. Use it inside of an exception handler to record that the exception occurred.

run

– (void)**run**

Starts the main event loop. The loop continues until a **stop:** or **terminate:** message is received. Upon each iteration through the loop, the next available event from the Window Server is stored and is then dispatched by sending the event to NSApp using **sendEvent:**.

Send a **run** message as the last statement from **main()**, after the application's objects have been initialized.

See also: – **applicationDidFinishLaunching:** (delegate method), – **runModalForWindow:**, – **runModalSession:**

runModalForWindow:

– (int)**runModalForWindow:**(NSWindow *)*aWindow*

Starts a modal event loop for *aWindow*. Until the loop is broken by a **stopModal**, **stopModalWithCode:**, or **abortModal** message, the application won't respond to any mouse, keyboard, or window-close events unless they're associated with *aWindow*. If **stopModalWithCode:** is used to stop the modal event loop, this method returns the argument passed to **stopModalWithCode:**. If **stopModal** is used, it returns the constant NSRunStoppedResponse. If **abortModal** is used, it returns the constant NSRunAbortedResponse. This method is functionally similar to the following:

```
NSModalSession session = [NSApp beginModalSessionforWindow:theWindow];

for (;;) {
 if ([NSApp runModalSession:session] != NSRunContinuesResponse)
  break;
}
[NSApp endModalSession:session];
```

See also: – **run**, – **runModalSession:**

runModalSession:

– (int)**runModalSession:**(NSModalSession)*session*

Runs a modal session represented by *session*, as defined in a previous invocation of **beginModalSessionForWindow:**. A loop using this method is similar to a modal event loop run with **runModalForWindow:**, except that with this method the application can continue processing between method invocations. When you invoke this method, events for the NSWindow of this session are dispatched as normal; this method returns when there are no more events. You must invoke this method frequently enough that the window remains responsive to events.

If the modal session was not stopped, this method returns NSRunContinuesResponse. If **stopModal** was invoked as the result of event processing, NSRunStoppedResponse is returned. If **stopModalWithCode:**

was invoked, this method returns the value passed to **stopModalWithCode:**. The NSAbortModalException raised by **abortModal** isn't caught, so **abortModal** will not stop the loop.

See also: – **endModalSession**, – **run**

runPageLayout:

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

Displays the application's page layout panel, an instance of NSPageLayout. If the NSPageLayout instance does not exist, it creates one. This method is typically invoked when the user selects Page Layout from the application's menu.

sendAction:to:from:

– (BOOL)**sendAction:**(SEL)*anAction* **to:**(id)*aTarget* **from:**(id)*sender*

Sends the message *anAction* to *aTarget*. If *aTarget is* **nil**, NSApp looks for an object that can respond to the message—that is, an object that implements a method matching *anAction*. It begins with the first responder of the key window. If the first responder can't respond, it tries the first responder's next responder and continues following next responder links up the responder chain. If none of the objects in the key window's responder chain can handle the message, NSApp attempts to send the message to the key window's delegate.

If the delegate doesn't respond and the main window is different from the key window, NSApp begins again with the first responder in the main window. If objects in the main window can't respond, NSApp attempts to send the message to the main window's delegate. If still no object has responded, NSApp tries to handle the message itself. If NSApp can't respond, it attempts to send the message to its own delegate.

Returns YES if the action is successfully sent; otherwise returns NO.

See also: – **targetForAction:**, – **tryToPerform:with:**, – **makeWindowsPerform:inOrder:**

sendEvent:

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

Dispatches *anEvent* to other objects. You rarely invoke **sendEvent:** directly although you might want to override this method to perform some action on every event. **sendEvent:** messages are sent from the main event loop (the **run** method). **sendEvent:** is the method that dispatches events to the appropriate responders; NSApp handles application events, the NSWindow indicated in the event record handles window related events, and mouse and key events are forwarded to the appropriate NSWindow for further dispatching.

See also: – **currentEvent**, – **postEvent:atStart:**

servicesMenu

– (NSMenu *)**servicesMenu**

Returns the Services menu or **nil** if no Services menu has been created.

See also: – **setServicesMenu:**

servicesProvider

– (NSMenu *)**servicesProvider**

Returns the object that provides the services that this application advertises in the Services menu of other applications.

See also: – **setServicesProvider:**

setApplicationIconImage:

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

Sets the application's icon to *anImage.*

See also: – **applicationIconImage**

setDelegate:

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

Makes *anObject* the receiver's delegate. The messages that a delegate can expect to receive are listed at the end of this specification. The delegate doesn't need to implement all the methods.

See also: – **delegate**

setMainMenu:

– (void)**setMainMenu:**(NSMenu *)*aMenu*

Makes *aMenu* the application's main menu.

See also: – **mainMenu**

setServicesMenu:

– (void)**setServicesMenu:**(NSMenu *)*aMenu*

Makes *aMenu* the application's Services menu.

See also: – **servicesMenu**

setServicesProvider:

– (void)**setServicesProvider:**(id)*aProvider*

Registers the object *aProvider* as the service provider. The service provider is an object that performs all of the services that the application provides to other applications. When another application requests a service from the receiver, it sends the service request to *aProvider*.

For more information on registering services, see the on-line document **/NextLibrary/Documentation/NextDev/TasksAndConcepts/ProgrammingTopics/Services.rtf**.

See also: – **servicesProvider**

setWindowsMenu:

– (void)**setWindowsMenu:**(NSMenu *)*aMenu*

Makes *aMenu* the application's Window menu.

See also: – **windowsMenu**

setWindowsNeedUpdate:

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

Sets whether the application's windows need updating when the application has finished processing the current event. This method is especially useful for making sure menus are updated to reflect changes not initiated by user actions, such as messages received from remote objects.

See also: – **updateWindows**

showHelp:

– (void)**showHelp:**(id)*sender*

Brings up the application's help file by sending a request to the shared NSWorkspace object to open the file using the default application for the help file's type. (You set the application's help file using Project

Builder.) This method is typically invoked when the user chooses the Help command or one of the commands from the Help menu.

On Microsoft Windows platforms, the help file is typically an HLP file, so this method brings up Microsoft Windows help. On Mach platforms, the help file is typically an RTF file and is displayed using Edit, but the help file can be anything. For example, Project Builder on Mach brings up a Digital Librarian bookshelf in response to its Help command.

For more information on providing on-line help for your application, see the NSHelpManager class specification.

See also: – **activateContextHelpMode:**

stop:

– (void)**stop:**(id)*sender*

Stops the main event loop. This method will break the flow of control out of the **run** method, thereby returning to the **main()** function. A subsequent **run** message will restart the loop.

If this method is invoked during a modal event loop, it will break that loop but not the main event loop.

See also: – **runModalForWindow:**, – **runModalSession:**, – **terminate:**

stopModal

– (void)**stopModal**

Stops a modal event loop. This method should always be paired with a previous invocation of **runModalForWindow:** or **beginModalSessionForWindow:**. When **runModalForWindow:** is stopped with this method, it returns NSRunStoppedResponse. This method will stop the loop only if it's executed by code responding to an event. If you need to stop a **runModalForWindow:** loop from a method registered with the current NSRunLoop (for example, a method repeatedly invoked by an NSTimer object), use the **abortModal** method.

See also: – **runModalSession:**, – **stopModalWithCode:**

stopModalWithCode:

– (void)**stopModalWithCode:**(int)*returnCode*

Like **stopModal**, except the argument *returnCode* allows you to specify the value that **runModalForWindow:** will return.

See also: – **abortModal**

targetForAction:

– (id)**targetForAction:**(SEL)*aSelector*

Returns the object that receives the action message *aSelector*.

See also: – **sendAction:to:from:**, – **tryToPerform:with:**

terminate:

– (void)**terminate:**(id)*sender*

Terminates the application. This method is typically invoked when the user chooses Quit or Exit from the application's menu. Each use of **terminate:** invokes **applicationShouldTerminate:** to notify the delegate that the application will terminate. If **applicationShouldTerminate:** returns NO, control is returned to the main event loop, and the application isn't terminated. Otherwise, this method posts an NSApplicationWillTerminateNotification to the default notification center. Don't put final cleanup code in your application's **main()** function; it will never be executed. If cleanup is necessary, have the delegate respond to **applicationWillTerminate:** and perform the cleanup in that method.

See also: – **run**, – **stop:**, **exit()**

tryToPerform:with:

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

Dispatches action messages. The receiver tries to perform the method *aSelector* using its inherited NSResponder method **tryToPerform:with:**. If the receiver doesn't perform *aSelector*, the delegate is given the opportunity to perform it using its inherited NSObject method **perform:withObject:**. If either the receiver or its delegate accept *aSelector*, this method returns YES; otherwise it returns NO.

See also: – **respondsToSelector:** (NSObject)

unhide:

– (void)**unhide:**(id)*sender*

Restores hidden windows to the screen and makes the application active. Invokes **unhideWithoutActivation**.

See also: – **activateIgnoringOtherApps:**, – **hide:**

unhideWithoutActivation

– (void)**unhideWithoutActivation**

Restores hidden windows without activating their owner (the receiver). When this method begins, it posts an NSApplicationWillUnhideNotification to the default notification center. If it completes successfully, it posts an NSApplicationDidUnhideNotification.

See also: – **activateIgnoringOtherApps:**, – **applicationDidUnhide:** (delegate method), – **applicationWillUnhide:** (delegate method), – **hide:**

updateWindows

– (void)**updateWindows**

Sends an **update** message to each on-screen NSWindow. This method is invoked automatically in the main event loop after each event. If the NSWindow has automatic updating turned on, its **update** method will redraw all of the NSWindow's NSViews that need redrawing. If automatic updating is turned off, the **update** message does nothing. (You turn automatic updating on and off by sending **setAutodisplay:** to an NSWindow.)

When this method begins, it posts an NSApplicationWillUpdateNotification to the default notification center. When it successfully completes, it posts an NSApplicationDidUpdateNotification.

See also: – **applicationWillUpdate:** (delegate method), – **applicationDidUpdate:** (delegate method), – **setWindowsNeedUpdate:**, – **setAutodisplay:** (NSWindow)

updateWindowsItem:

– (void)**updateWindowsItem:**(NSWindow *)*aWindow*

Updates the Window menu item for *aWindow* to reflect the edited status of *aWindow*. You rarely need to invoke this method because it is invoked automatically when the edit status of an NSWindow is set.

See also: – **changeWindowsItem:title:filename:**, – **setDocumentEdited:** (NSWindow)

validRequestorForSendType:returnType:

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

Indicates whether the receiver can send and receive the specified pasteboard types. This message is sent to all responders in a responder chain. NSApp is typically the last item in the responder chain, so it usually only receives this message if none of the current responders can send *sendType* data and accept back *returnType* data.

The receiver passes this message on to its delegate if the delegate can respond (and isn't an NSResponder with its own next responder). If the delegate can't respond or returns **nil**, this method returns **nil**. If the delegate can find an object that can send *sendType* data and accept back *returnType* data, that object is returned.

See also: – **validRequestorForSendType:returnType:** (NSResponder),
– **registerServicesMenuSendTypes:andReturnTypes:**,
– **readSelectionFromPasteboard:** (NSServicesRequests protocol),
– **writeSelectionToPasteboard:types:** (NSServicesRequests protocol)

windowWithWindowHandle:

– (NSWindow *)**windowWithWindowHandle:**(void *)*hWnd*

On Microsoft Windows platforms, returns the NSWindow object associated with the Win32 window handle *hWnd*. If the application does not own *hWnd* or *hWnd* does not have an NSWindow associated with it, this method returns **nil**. This method is for Microsoft Windows platforms only. **windowWithWindowHandle:** is not defined for the Mach platform.

See also: – **windowWithWindowHandle:** (NSWindow), – **windowHandle** (NSWindow)

windowWithWindowNumber:

– (NSWindow *)**windowWithWindowNumber:**(int)*windowNum*

Returns the NSWindow object corresponding to *windowNum*.

windows

– (NSArray *)**windows**

Returns an NSArray of the application's NSWindows, including off-screen windows.

windowsMenu

– (NSMenu *)**windowsMenu**

Returns the Window menu or **nil** if no Window menu has been created.

See also: – **setWindowsMenu:**

Notifications

NSApplicationDidBecomeActiveNotification

Posted immediately after the application becomes active. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationDidFinishLaunchingNotification

Posted at the end of the **finishLaunching** method to indicate that the application has completed launching and is ready to run.

The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationDidHideNotification

Posted at the end of the **hide:** method to indicate that the application is now hidden. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationDidResignActiveNotification

Posted immediately after the application gives up its active status to another application. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationDidUnhideNotification

Posted at the end of the **unhideWithoutActivation** method to indicate that the application is now visible. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationDidUpdateNotification

Posted at the end of the **updateWindows** method to indicate that the application has finished updating its windows. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillBecomeActiveNotification

Posted immediately after the application becomes active. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillFinishLaunchingNotification

Posted at the start of the **finishLaunching** method to indicate that the application has completed its initialization process and is about to finish launching. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillHideNotification

Posted at the start of the **hide:** method to indicate that the application is about to be hidden. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillResignActiveNotification

Posted immediately before the application gives up its active status to another application. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillTerminateNotification

Posted by the **terminate:** method to indicate that the application will terminate. Posted only if the delegate method **applicationShouldTerminate:** returns YES. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillUnhideNotification

Posted at the start of the **unhideWithoutActivation** method to indicate that the application is about to be visible. The notification contains:

Notification Object	NSApp
Userinfo	None

NSApplicationWillUpdateNotification

Posted at the start of the **updateWindows** method to indicate that the application is about to update its windows. The notification contains:

Notification Object	NSApp
Userinfo	None

Methods Implemented By the Delegate

application:openFile:

– (BOOL)**application:**(NSApplication *)*theApplication* **openFile:**(NSString *)*filename*

Sent directly by *theApplication* to the delegate. The method should open the file *filename*, returning YES if the file is successfully opened, and NO otherwise.

Note: If the user has started up the application by double-clicking a file, the delegate receives the **application:openFile:** message before receiving **applicationDidFinishLaunching:**. (**applicationWillFinishLaunching:** is sent before **application:openFile:**.)

See also: – **application:openFileWithoutUI:**, – **application:openTempFile:**, – **applicationOpenUntitledFile:**

application:openFileWithoutUI:

– (BOOL)**application:**(NSApplication *)*sender* **openFileWithoutUI:**(NSString *)*filename*

Sent directly by *sender* to the delegate to request that the file *filename* be opened as a linked file. The method should open the file without bringing up its application's user interface; that is, work with the file is under programmatic control of *sender*, rather than under keyboard control of the user. Returns YES if the file was successfully opened, NO otherwise.

See also: – **application:openFile:**, – **appliction:openTempFile:**, – **application:printFile:**, – **applicationOpenUntitledFile:**

application:openTempFile:

– (BOOL)**application:**(NSApplication *)*theApplication* **openTempFile:**(NSString *)*filename*

Sent directly by *theApplication* to the delegate. The method should attempt to open the file *filename*, returning YES if the file is successfully opened, and NO otherwise.

By design, a file opened through this method is assumed to be temporary; it's the application's responsibility to remove the file at the appropriate time.

See also: – **application:openFile:**, – **application:openFileWithoutUI:**, – **applicationOpenUntitledFile:**

application:printFile:

– (BOOL)**application:**(NSApplication *)*theApplication* **printFile:**(NSString *)*filename*

Sent when the user starts up the application on the command line with the **-NSPrint** option. Sent directly by *theApplication* to the delegate.

The method should attempt to print the file *filename*, returning YES if the file was successfully printed, and NO otherwise. The application terminates (using the **terminate:** method) after this method returns.

If at all possible, this method should print the file without displaying the user interface. For example, if you pass the **-NSPrint** option to the TextEdit application, TextEdit assumes you want to print the entire contents

of the specified file. However, if the application opens more complex documents, you may want to display a panel that lets user choose exactly what they want to print.

See also: – **application:openFileWithoutUI:**

applicationDidBecomeActive:

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

Sent by the default notification center immediately after the application becomes active. *aNotification* is always an NSApplicationDidBecomeActiveNotification. You can retrieve the NSApplication object by sending the **object** method to *aNotification*.

See also: – **applicationDidFinishLaunching:**, – **applicationDidResignActive:**, – **applicationWillBecomeActive:**

applicationDidFinishLaunching:

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

Sent by the default notification center after the application has been launched and initialized but before it has received its first event. *aNotification* is always an NSApplicationDidFinishLaunchingNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*. The delegate can implement this method to perform further initialization.

Note: If the user has started up the application by double-clicking a file, the delegate receives the **application:openFile:** message before receiving **applicationDidFinishLaunching:**. (**applicationWillFinishLaunching:** is sent before **application:openFile:**.)

See also: – **applicationDidBecomeActive:**, – **applicationWillFinishLaunching:**, – **finishLaunching**

applicationDidHide:

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

Sent by the default notification center immediately after the application is hidden. *aNotification* is always an NSApplicationDidHideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification.*

See also: – **applicationWillHide:**, – **applicationDidUnhide:**, – **hide**:

applicationDidResignActive:

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

Sent by the default notification center immediately after the application is deactivated. *aNotification* is always an NSApplicationDidResignActiveNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidBecomeActive:**, – **applicationWillResignActive:**

applicationDidUnhide

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

Sent by the default notification center immediately after the application is made visible. *aNotification* is always an NSApplicationDidUnhideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification.*

See also: – **applicationDidHide:**, – **applicationWillUnhide:**, – **unhide**:

applicationDidUpdate:

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

Sent by the default notification center immediately after the NSApplication object updates its NSWindows. *aNotification* is always an NSApplicationDidUpdateNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationWillUpdate:**, – **updateWindows**

applicationOpenUntitledFile:

– (BOOL)**applicationOpenUntitledFile:**(NSApplication *)*theApplication*

Sent directly by *theApplication* to the delegate to request that a new, untitled file be opened. Returns YES if the file was successfully opened, NO otherwise.

See also: – **application:openFile:**, – **application:openFileWithoutUI:**, – **application:openTempFile:**

applicationShouldTerminate:

– (BOOL)**applicationShouldTerminate:**(NSApplication *)*sender*

Invoked from within the **terminate:** method immediately before the application terminates. *sender* is the NSApplication to be terminated. If this method returns NO, the application is not terminated, and control returns to the main event loop. Return YES to allow the application to terminate.

See also: – **applicationShouldTerminateAfterLastWindowClosed:**, – **applicationWillTerminate:**, – **terminate:**

applicationShouldTerminateAfterLastWindowClosed:

– (BOOL)**applicationShouldTerminateAfterLastWindowsClosed:**(NSApplication *)*theApplication*

Invoked when the user closes the last window that the application has open on.

This method is intended for the Microsoft Windows platform. On Microsoft Windows, the default behavior is to terminate the application if the user closes the last window. Most application use this default behavior; however, you may choose to have **applicationShouldTerminateAfterLastWindowClosed:** perform some other function, such as display a panel that gives the user a choice of exiting the application or opening a new window.

If this method returns NO, the application is not terminated, and control returns to the main event loop. Return YES to allow the application to terminate. Note that **applicationShouldTerminate:** is invoked if this method returns YES.

See also: – **terminate:**

applicationWillBecomeActive:

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

Sent by the default notification center immediately before the application becomes active. *aNotification* is always an NSApplicationWillBecomeActiveNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationDidBecomeActive:**, – **applicationWillFinishLaunching:**, – **applicationWillResignActive:**

applicationWillFinishLaunching:

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

Sent by the default notification center immediately before the NSApplication object is initialized. *aNotification* is always an NSApplicationWillFinishLaunchingNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification.*

See also: – **applicationDidFinishLaunching:**, – **applicationWilBecomeActive:**, – **finishLaunching**

applicationWillHide:

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

Sent by the default notification center immediately after the application is hidden. *aNotification* is always an NSApplicationWillHideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification.*

See also: – **applicationDidHide:**, – **hide**:

applicationWillTerminate:

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

Sent by the default notification center immediately before the application terminates. *aNotification* is always an NSApplicationWillTerminateNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*. Put any necessary cleanup code in this method.

See also: – **applicationShouldTerminate:**, – **terminate:**

applicationWillResignActive:

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

Sent by the default notification center immediately after the application is deactivated. *aNotification* is always an NSApplicationWillResignActiveNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification*.

See also: – **applicationWillBecomeActive:**, – **applicationDidResignActive:**

applicationWillUnhide

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

Sent by the default notification center immediately after the application is unhidden. *aNotification* is always an NSApplicationWillUnhideNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification.*

See also: – **applicationDidUnhide:**, – **applicationWillHide:**, – **unhide**:

applicationWillUpdate:

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

Sent by the default notification center immediately before the NSApplication object updates its NSWindows. *aNotification* is always an NSApplicationWillUpdateNotification. You can retrieve the NSApplication object in question by sending **object** to *aNotification.*

See also: – **applicationDidUpdate:**, – **updateWindows**