OPENSTEP™ SPECIFICATION

October 19, 1994

This document sets forth the OpenStep application programming interface (API).

Contents

Introduction

This document describes the application programming interface (API) of OpenStep™. OpenStep is an operating system independent, object-oriented application layer, based on NeXT's advanced object technology. OpenStep contains these major components:

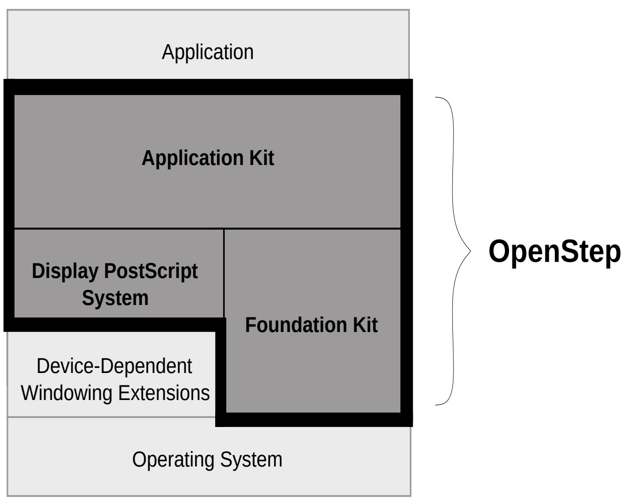

Figure 1. Major Components of OpenStep

Application Kit The Application Kit™ provides the basic software for writing interactive applications—applications that use windows, draw on the screen, and respond to user actions on the keyboard and mouse. The Application Kit contains the components that define the OpenStep user interface.

Foundation Kit The Foundation Kit™ provides the fundamental building blocks that applications use to manage data and resources. It defines facilities for handling multibyte character sets, object persistency and distribution, and provides an interface to common operating system facilities.

Display PostScript System The Display PostScript® system provides OpenStep with its device-independent imaging model.

The OpenStep API is expressed in the Objective C language, an object-oriented extension of ANSI C. The language itself lies outside of the scope of this specification. For information on Objective C, see *NEXTSTEP Object-Oriented Programming and the Objective C Language* (Addison-Wesley Publishing Co., 1993). Please note that many of the types used for method argument and return values in the OpenStep specification are defined in the Objective C language. These include:

BOOL
Class
id
IMP
nil
Protocol
SEL

In addition, the type codes used to encode method argument and return types for archiving and other purposes are also defined in the Objective C language.

How this Document Is Organized

The three components of OpenStep are described in separate chapters of this document, starting with Chapter 1, "The Application Kit". Each chapter is organized in the same way, having these standard sections:

Classes

This section lists the API for each class defined in the component. For each class, these subsections may appear:

Inherits From: The inheritance hierarchy for the class. For example:

NSPanel : NSWindow : NSResponder : NSObject

The first class listed (NSPanel, in this example) is the class's superclass. The last class listed is generally NSObject, the root of almost all OpenStep inheritance hierarchies. The classes between show the chain of inheritance from NSObject to the superclass. (This particular example shows the inheritance hierarchy for the NSMenu class of the Application Kit.)

Conforms To: The formal protocols that the class conforms to. These include both protocols the class adopts and those it inherits from other adopting classes. If inherited, the name of the adopting class in given in parentheses. For example:

NSCoding
NSCopying
NSMutableCopying
NSObject (NSObject)

(This particular example is from the NSArray class in the Foundation Kit.)

Declared In: The header file that declares the class interface. For example:

Foundation/NSString.h

(This example is from the NSString class.)

Next, the methods the class declares and implements are listed by name and grouped by type. For example, methods used to draw are listed separately from methods used to handle events. This listing includes all the methods declared in the class. It also may include a method declared in a protocol the class conforms to, if there is something extraordinary about the class's implementation of the method. Each method is accompanied by a brief description which states what the method does and mentions the arguments and return value, if any.

If a class lets you define another object—a delegate—that can intercede on behalf of instances of the class, the methods that the delegate can implement are described in a separate section. These are not methods defined in the class; rather, they're methods that you can define to respond to messages sent from instances of the class. In essence, this section documents an informal protocol. But because these methods are so closely tied to the behavior of a particular class, they're documented with the class rather than in the "Protocols" section.

Some class specifications have separate sections with titles such as "Methods Implemented by the Superview", "Methods Implemented by Observers", or "Methods Implemented by the Owner." These are also informal protocols. They document methods that can or must be implemented to receive messages on behalf of instances of the class.

Protocols

The protocols section documents both formal and informal protocols. Formal protocols are those that are declared using the **@protocol** compiler directive. They can be formally adopted and implemented by a class and tested by sending an object a **conformsToProtocol:** message.

Some formal protocols are adopted and implemented by OpenStep classes. However, many formal protocols are declared by a kit, but not implemented by it. They list methods that you can implement to respond to kit-generated messages.

A few formal protocols are implemented by a kit, but not by a class that's part of the documented API. Rather, the protocol is implemented by an anonymous object that the kit supplies. The protocol lets you know what messages you can send to the object.

Like formal protocols, informal protocols declare a list of methods that others are invited to implement. If an informal protocol is closely associated with one particular class—for example, the list of methods implemented by the delegate—it's documented in the class description. Informal protocols associated with more than one class, or not associated with any particular class, are documented with the formal protocols in this section.

Protocol information is organized into many of the same sections as described above for a class specification. But protocols are not classes and therefore differ somewhat in the kind of information provided. The sections of a protocol specification are shown in bold in the following:

Adopted By: A list of the OpenStep classes that adopt the protocol. Many protocols declare methods that applications must implement and so are not adopted by any OpenStep classes.

Some protocols are implemented by anonymous objects (instances of an unknown class); the protocol is the only information available about what messages the object can respond to. Protocols that have an implementation available through an anonymous object generally don't have to be reimplemented by other classes.

An informal protocol can't be formally adopted by a class and it can't formally incorporate another protocol. So its description begins with information about the category where it's declared:

Category Of: The class that the category belongs to. Informal protocols are typically declared as categories of the NSObject class. This gives them the widest possible scope.

All descriptions of protocols, whether formal or informal, list where the protocol is declared:

Declared In: The header file where the protocol is declared.

If the protocol includes enough methods to warrant it, they're divided by type and presented just as the methods of a class are.

Functions

Related functions are grouped together under a heading that describes the common purpose. Each function, its arguments, and its return value are briefly described in an accompanying comment.

Types and Constants

Related defined types, enumeration constants, symbolic constants, structures, and global variables are grouped together under a heading that describes the common purpose. A short description accompanies each group.

1 *Application Kit*

Introduction

The Application Kit defines Objective C classes, protocols, C functions, constants, and data types that are designed to be used by virtually every OpenStep application. The principal aim of the Application Kit is to provide the framework for implementing a graphical, event-driven application.

Classes

The Application Kit contains over sixty classes which inherit directly or indirectly from NSObject, the root class defined in the Foundation Kit. The following diagram shows the inheritance relationship among these classes. After the diagram, the specifications for these classes are arranged in alphabetical order.

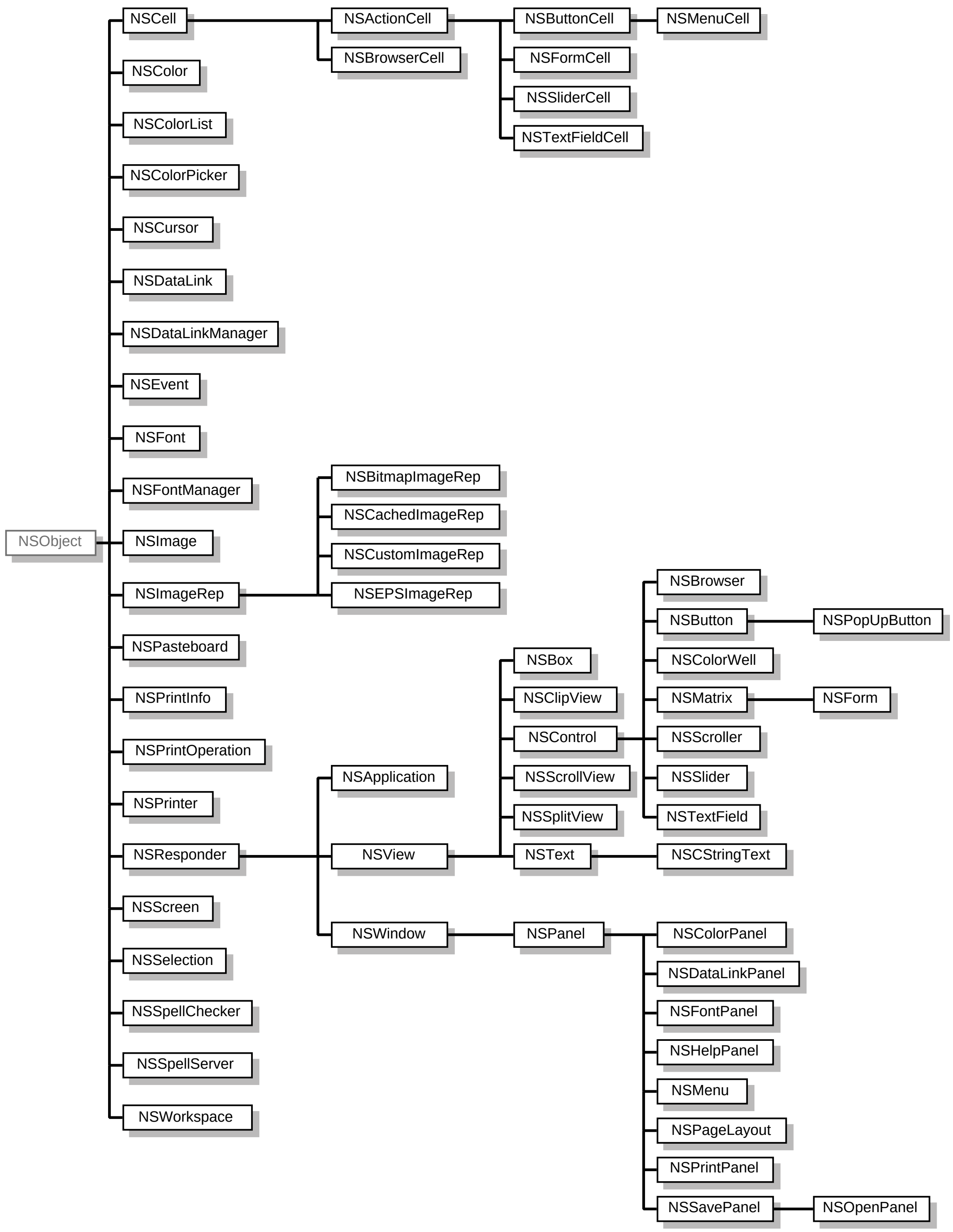

Figure 1-1. Application Kit Classes

NSActionCell

Inherits From: NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSActionCell.h

Class Description

An NSActionCell defines an active area inside a control (an instance of NSControl or one of its subclasses). As an NSControl's active area, an NSActionCell does three things: it usually performs display of text or an icon (the subclass NSSliderCell is an exception); it provides the NSControl with a target and an action; and it handles mouse (cursor) tracking by properly highlighting its area and sending action messages to its target based on cursor movement. The only way to specify the NSControl for a particular NSActionCell is to send the NSActionCell a **drawWithFrame:inView:** message, passing the NSControl as the argument for the **inView:** keyword of the method.

NSActionCell implements the target object and action method as defined by its superclass, NSCell. As a user manipulates an NSControl, NSActionCell's **trackMouse:inRect:ofView:untilMouseUp:** method (inherited from NSCell) updates its appearance and sends the action message to the target object with the NSControl object as the only argument.

Usually, the responsibility for an NSControl's appearance and behavior is completely given over to a corresponding NSActionCell. (NSMatrix, and its subclass NSForm, are NSControls that don't follow this rule.)

A single NSControl may have more than one NSActionCell. To help identify it in this case, every NSActionCell has an integer tag. Note, however, that no checking is done by the NSActionCell object itself to ensure that the tag is unique. See the NSMatrix class for an example of a subclass of NSControl that contains multiple NSActionCells.

Many of the methods that define the contents and look of an NSActionCell, such as **setFont:** and **setBordered:**, are reimplementations of methods inherited from NSCell. They're subclassed to ensure that the NSActionCell is redisplayed if it's currently in an NSControl.

Configuring an NSActionCell

– (void)**setAlignment:**(NSTextAlignment)*mode*	Sets the NSActionCell's text alignment to *mode*.
– (void)**setBezeled:**(BOOL)*flag*	Adds or removes the NSActionCell's bezel.
– (void)**setBordered:**(BOOL)*flag*	Adds or removes the NSActionCell's border.
– (void)**setEnabled:**(BOOL)*flag*	Sets whether the NSActionCell reacts to mouse and keyboard events.

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

Sets the NSActionCell's floating point format.

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

Sets the NSActionCell's font to *fontObject*.

– (void)**setImage:**(NSImage *)*image*

Sets the NSActionCell's icon to *image.*

Manipulating NSActionCell Values

– (double)**doubleValue**

Returns the NSActionCell's contents as a **double**.

– (float)**floatValue**

Returns the NSActionCell's contents as a **float**.

– (int)**intValue**

Returns the NSActionCell's contents as an **int**.

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

Sets the NSActionCell's contents to a copy of *aString*.

– (NSString *)**stringValue**

Returns the NSActionCell's contents as a string.

Displaying

– (void)**drawWithFrame:**(NSRect)*cellFrame*
inView:(NSView *)*controlView*

Draws the NSActionCell in the rectangle *cellFrame* of *controlView* (which should normally be an NSControl).

– (NSView *)**controlView**

Returns the view (normally an NSControl) in which the NSActionCell was last drawn.

Target and Action

– (SEL)**action**

Returns the NSActionCell's action method.

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

Sets the NSActionCell's action method to *aSelector.*

– (void)**setTarget:**(id)*anObject*

Sets the NSActionCell's target object to *anObject*.

– (id)**target**

Returns the NSActionCell's target object.

Assigning a Tag

– (void)**setTag:**(int)*anInt*

Sets the NSActionCell's tag to *anInt*.

– (int)**tag**

Returns the NSActionCell's tag.

NSApplication

Inherits From: NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

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

Class Description

The NSApplication class provides the central framework of your application's execution. Every application must have exactly one instance of NSApplication (or of a custom subclass of NSApplication). Your program's **main()** function should create this instance by calling the **sharedApplication** class method. (Alternatively, you could use **alloc** and **init**, making sure they're called only once.) After creating the NSApplication, the **main()** function should load your application's main nib file, and then start the event loop by sending the NSApplication a **run** message. Here's an example of a typical OpenStep **main()** function in its entirety:

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

Creating the NSApplication object connects the program to the window system and the Display PostScript server, and initializes its PostScript environment. The NSApplication object maintains a list of all the NSWindows that the application uses, so it can retrieve any of the application's NSViews.

The NSApplication object's main task is to receive events from the window system and distribute them to the proper NSResponders. The NSApplication translates an event into an NSEvent object, then forwards the NSEvent to the affected NSWindow object. A key-down event that occurs while the Command key is pressed results in a **commandKey:** message, and every NSWindow has an opportunity to respond to it. Other keyboard and mouse events are sent to the NSWindow associated with the event; the NSWindow then distributes these NSEvents to the objects in its view hierarchy.

In general, it's neater and cleaner 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 the NSApplication object. It's rarely necessary to create a custom subclass of NSApplication. You will need to do so only if you need to provide your own special response to messages that are routinely sent to the NSApplication object. To use a custom subclass of NSApplication, simply substitute it for NSApplication in the **main()** function above.

When you create an instance of NSApplication (or of a custom subclass of NSApplication), it gets stored as the global variable NSApp. Although this global variable isn't used in the example **main()** function above, you might find it convenient to refer to NSApp within the source code for your application's custom objects. Note that you can also retrieve the NSApplication object by invoking **sharedApplication**.

The NSApplication class sets up autorelease pools during initialization and during the event loop—that is, 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. The autorelease pools aren't accessible outside the scope of the respective NSApplication and NSBundle methods. This isn't usually a problem, because a typical OpenStep application instantiates its objects by loading nib files (and by having the objects from the nib file create other objects during initialization and during the event loop). However, if you do need to use OpenStep classes within the **main()** function itself (other than to invoke the methods just mentioned), you should instantiate an autorelease pool before using the classes, and then release the pool once you're done. For more information, see the description of the NSAutoreleasePool class in the Foundation Kit.

The Delegate and Observers

The NSApplication object can be assigned a delegate that responds on behalf of the NSApplication to certain messages addressed to the NSApplication object. Some of these messages, such as **application:openFile:withType:**, ask the delegate to open a file. Another message, **applicationShouldTerminate:**, lets the delegate determine whether the application should be allowed to quit.

An NSApplication can also have *observers*. Observers receive notifications of changes in the NSApplication, but they don't have the unique responsibility that a delegate has. Any instance of a class that implements an observer method can register to receive the corresponding notification. For example, if a class implements **applicationDidFinishLaunching:** and registers to receive the corresponding notification, instances of this class are given an opportunity to react after the NSApplication has been initialized. (The observer methods are listed later in this class specification. For information about how to register to receive notifications, see the class specification for the Foundation Kit's NSNotificationCenter class.)

There can be only one delegate, but there can be many observers. The delegate itself can be an observer—in fact, in many applications the delegate might be the only observer. Whereas most observers need to explicitly register with an NSNotificationCenter before they can receive a particular notification message, the delegate need only implement the method. By simply implementing an observer method, the NSApplication's delegate is automatically registered to receive the corresponding notification.

Creating and Initializing the NSApplication

+ (NSApplication *)**sharedApplication**	Returns the NSApplication instance, creating it if it doesn't yet exist.

– (void)**finishLaunching**

Activates the application, opens any files specified by the "NSOpen" user default, and unhighlights the application's icon in the Workspace Manager. This method is invoked by **run** before it starts the event loop. When this method begins, it posts the notification NSApplicationWillFinishLaunchingNotification with the receiving object to the default notification center. When it successfully completes, it posts the notification NSApplicationDidFinishLaunchingNotification. If you override **finishLaunching**, the subclass method should invoke the superclass method.

Changing the Active Application

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

Makes this the active application. If *flag* is NO, the application is activated only if no other application is currently active.

– (void)**deactivate**

Deactivates the application.

– (BOOL)**isActive**

Returns whether this is the active application.

Running the Event Loop

– (void)**abortModal**

Aborts the event loop started by **runModalForWindow:**.

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

Sets up a modal session with *theWindow.*

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

Finishes a modal session.

– (BOOL)**isRunning**

Returns whether the main event loop is running.

– (void)**run**

Starts the main event loop.

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

Starts a modal event loop for *theWindow.*

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

Runs a modal session.

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

Dispatches events to other objects. When sending the activate application event, this method posts the notifications NSApplicationWillBecomeActive and NSApplicationDidBecomeActive with the receiving object to the default notification center. When sending the deactivate application event, it posts the NSApplicationWillResignActiveNotification and NSApplicationDidResignActiveNotification notifications with the receiving object to the default notification center.

– (void)**stop:**(id)*sender*

Stops the main event loop.

– (void)**stopModal**

Stops the modal event loop.

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

Stops the event loop started by **runModalForWindow:** and sets the code that **runModalForWindow:** will return.

Getting, Removing, and Posting Events

– (NSEvent *)**currentEvent**

Returns the current event.

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

Removes from the event queue all events matching *mask* that were generated before *lastEvent.*

– (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. The *mode* argument names an NSRunLoop mode that determines what other ports are listened to and what timers may fire while the NSApplication is waiting for the event.

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

Adds *event* to the beginning of the application's event queue if *flag* is YES, and to the end otherwise.

Sending Action Messages

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

Sends an action message to *aTarget* or up the responder chain.

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

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

– (BOOL)**tryToPerform:**(SEL)*aSelector* **with:**(id)*anObject*
Attempts to send a message to the application or the delegate.

Setting the Application's Icon

– (void)**setApplicationIconImage:**(NSImage *)*anImage*
Sets the application's icon to *anImage.*

– (NSImage *)**applicationIconImage**
Returns the NSImage used for the application's icon.

Hiding All Windows

– (void)**hide:**(id)*sender*
Hides all the application's windows. When this method begins, it posts the notification NSApplicationWillHideNotification with the receiving object to the default notification center. When it completes successfully, it posts the notification NSApplicationDidHideNotification.

– (BOOL)**isHidden**
Returns YES if windows are hidden.

– (void)**unhide:**(id)*sender*
Restores hidden windows to the screen.

– (void)**unhideWithoutActivation**
Restores hidden windows without activating their owner. When this method begins, it posts the notification NSApplicationWillUnhideNotification with the receiving object to the default notification center. When it completes successfully, it posts the notification NSApplicationDidUnhideNotification.

Managing Windows

– (NSWindow *)**keyWindow**
Returns the key window.

– (NSWindow *)**mainWindow**
Returns the main window.

– (NSWindow *)**makeWindowsPerform:**(SEL)*aSelector* **inOrder:**(BOOL)*flag*
Sends the *aSelector* message to the application's NSWindows—in front-to-back order if *flag* is YES, otherwise in the order of the array that the **windows** method returns.

– (void)**miniaturizeAll:**(id)*sender*
Miniaturizes all the receiver's application windows.

– (void)**preventWindowOrdering**
Suppresses the usual window ordering in handling the most recent mouse-down event.

– (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.

– (void)**updateWindows**

Sends an **update** message to on-screen NSWindows. When this method begins, it sends the notification NSApplicationWillUpdateNotification with the receiving object to the default notification center. When it successfully completes, it sends the notification NSApplicationDidUpdateNotification.

– (NSArray *)**windows**

Returns an array of the application's NSWindows.

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

Returns the NSWindow object corresponding to *windowNum*.

Showing Standard Panels

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

Brings up the color panel.

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

Shows the shared instance of the data link panel, creating it first if necessary.

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

Shows the application's help panel or the default one.

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

Runs the application's page layout panel.

Getting the Main Menu

– (NSMenu *)**mainMenu**

Returns the **id** of the application's main menu.

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

Makes *aMenu* the application's main menu.

Managing the Windows Menu

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

Adds a Windows menu item for *aWindow.*

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

Orders all registered NSWindows to the front.

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

Changes the Windows menu item for *aWindow.*

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

Removes the Windows menu item for *aWindow.*

– (void)**setWindowsMenu:**(id)*aMenu* Sets the Windows menu.

– (void)**updateWindowsItem:**(id)*aWindow* Updates the Windows menu item for *aWindow.*

– (NSMenu *)**windowsMenu** Returns the Windows menu.

Managing the Services menu

– (void)**registerServicesMenuSendTypes:**(NSArray *)*sendTypes*
returnTypes:(NSArray *)*returnTypes* Registers pasteboard types the application can send and receive.

– (NSMenu *)**servicesMenu** Returns the Services menu.

– (void)**setServicesMenu:**(NSMenu *)*aMenu* Sets the Services menu.

– (id)**validRequestorForSendType:**(NSString *)*sendType*
returnType:(NSString *)*returnType* Indicates whether the NSApplication can send and receive the specified types.

Getting the Display PostScript Context

– (NSDPSContext *)**context** Returns the NSApplication's Display PostScript context.

Reporting an Exception

– (void)**reportException:**(NSException *)*anException*
Logs the given exception by calling **NSLog()**.

Terminating the Application

– (void)**terminate:**(id)*sender* Frees the NSApplication object and exits the application.

Assigning a Delegate

– (id)**delegate** Returns the NSApplication's delegate.

– (void)**setDelegate:**(id)*anObject* Makes *anObject* the NSApplication's delegate.

Implemented by the Delegate

– (BOOL)**application:**(id)*sender*
openFileWithoutUI:(NSString *)*filename* Sent directly by *sender* to the delegate. Opens the specified file to run without a user interface.Work with the file will be under programmatic control of *sender*, rather than under keyboard control of the user. Returns YES or NO to indicate whether the file was successfully opened

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

Sent directly by *application* to the delegate. Like **application:openFileWithoutUI:**, but brings up the user interface of the file's application.

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

Sent directly by *application* to the delegate. Like **application:openFile:**, but 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.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationDidBecomeActiveNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationDidFinishLaunchingNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationDidHideNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationDidResignActiveNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationDidUnhideNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationDidUpdateNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent directly by *application* to the delegate. Like **application:openFile:**, but opens a new, untitled document.

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

Sent directly by *sender* to the delegate. Returns YES if the application should terminate.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationWillBecomeActiveNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationWillFinishLaunchingNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationWillHideNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationWillResignActiveNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationWillUnhideNotification. If the delegate implements this method, it's automatically registered to receive the notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSApplicationWillUpdateNotification. If the delegate implements this method, it's automatically registered to receive this notification.

NSBitmapImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding, NSCopying (NSImageRep)
NSObject (NSObject)

Declared In: AppKit/NSBitmapImageRep.h

Class Description

An NSBitmapImageRep is an object that can render an image from bitmap data. The data can be in Tag Image File Format (TIFF), or it can be raw image data. If it's raw data, the object must be informed about the structure of the image—its size, the number of color components, the number of bits per sample, and so on—when it's first initialized. If it's TIFF data, the object can get this information from the various TIFF fields included with the data.

Although NSBitmapImageReps are often used indirectly, through instances of the NSImage class, they can also be used directly—for example to manipulate the bits of an image as you might need to do in a paint program.

Setting Up an NSBitmapImageRep

A new NSBitmapImageRep is passed bitmap data for an image when it's first initialized. An NSBitmapImageRep can also be created from bitmap data that's read from a specified rectangle of a focused NSView.

Although the NSBitmapImageRep class inherits NSImageRep methods that set image attributes, these methods shouldn't be used. Instead, you should either allow the object to find out about the image from the TIFF fields or use methods defined in this class to supply this information when the object is initialized.

TIFF Compression

TIFF data can be read and rendered after it has been compressed using any one of the four schemes briefly described below:

LZW	Compresses and decompresses without information loss, achieving compression ratios up to 5:1. It may be somewhat slower to compress and decompress than the PackBits scheme.
PackBits	Compresses and decompresses without information loss, but may not achieve the same compression ratios as LZW.
JPEG	Compresses and decompresses with some information loss, but can achieve compression ratios anywhere from 10:1 to 100:1. The ratio is determined by a user-settable factor ranging from 1.0 to 255.0, with higher factors yielding greater compression. More information is lost with greater compression, but 15:1 compression is safe for publication quality. Some images can be compressed even more. JPEG compression can be used only for images that specify at least 4 bits per sample.
CCITTFAX	Compresses and decompresses 1 bit grayscale images using international fax compression standards CCITT3 and CCITT4.

An NSBitmapImageRep can also produce compressed TIFF data for its image using any of these schemes.

Allocating and Initializing a New NSBitmapImageRep Object

+ (id)**imageRepWithData:**(NSData *)*tiffData* — Creates and returns an initialized NSBitmapImageRep corresponding to the first image in *tiffData*.

+ (NSArray *)**imageRepsWithData:**(NSData *)*tiffData* — Creates and returns initialized NSBitmapImageRep objects for all the images in *tiffData*.

– (id)**initWithData:**(NSData *)*tiffData* — Initializes a newly allocated NSBitmapImageRep from the first TIFF header and image data found in *tiffData.*

– (id)**initWithFocusedViewRect:**(NSRect)*rect* — Initializes the new object using data read from the image contained in the rectangle *rect*.

– (id)**initWithBitmapDataPlanes:**(unsigned char **)*planes*
pixelsWide:(int)*width*
pixelsHigh:(int)*height*
bitsPerSample:(int)*bps*
samplesPerPixel:(int)*spp*
hasAlpha:(BOOL)*alpha*
isPlanar:(BOOL)*config*
colorSpaceName:(NSString *)*colorSpaceName*
bytesPerRow:(int)*rowBytes*
bitsPerPixel:(int)*pixelBits*

Initializes the new object from raw bitmap data in the *planes* data buffers. As the data is raw, the other arguments specify its attributes.

Getting Information about the Image

– (int)**bitsPerPixel**
Returns how many bits are needed to specify one pixel.

– (int)**samplesPerPixel**
Returns the number of samples (components) in the data.

– (BOOL)**isPlanar**
Returns YES if in planar configuration, NO if meshed.

– (int)**numberOfPlanes**
Returns the number of data planes.

– (int)**bytesPerPlane**
Returns the number of bytes in each data plane.

– (int)**bytesPerRow**
Returns the number of bytes in a scan line.

Getting Image Data

– (unsigned char *)**bitmapData**
Returns a pointer to the bitmap data. If the data is planar, returns a pointer to the first plane.

– (void)**getBitmapDataPlanes:**(unsigned char **)*data*
Provides pointers to each plane of bitmap data.

Producing a TIFF Representation of the Image

+ (NSData *)**TIFFRepresentationOfImageRepsInArray:**(NSArray *)*anArray*
Returns a TIFF representation of the images in the specified NSArray, using the compression that's returned by **getCompression:factor:** (if applicable).

+ (NSData *)**TIFFRepresentationOfImageRepsInArray:**(NSArray *)*anArray*
usingCompression:(NSTIFFCompression)*compressionType*
factor:(float)*factor*
Returns a TIFF representation of the images in the specified NSArray, which are compressed using *compressionType* and *factor*. If the specified compression isn't applicable, no compression is used.

– (NSData *)**TIFFRepresentation** Returns a TIFF representation of the image, using the compression that's returned by **getCompression:factor:** (if applicable).

– (NSData *)**TIFFRepresentationUsingCompression:**(NSTIFFCompression)*compressionType*
factor:(float)*factor* Returns a compressed TIFF representation of the image, having the specified compression type and compression factor. If the specified compression isn't applicable, no compression is used. Raises NSTIFFException if an atempt is made to create a TIFF representation using OpenStep custom color space bitmaps.

Setting and Checking Compression Types

+ (void)**getTIFFCompressionTypes:**(const NSTIFFCompression **)*list*
count:(int *)*numTypes* Returns all available compression types.

+ (NSString *)**localizedNameForTIFFCompressionType:**(NSTIFFCompression)*compression*
Returns the localized name for the compression type.

– (BOOL)**canBeCompressedUsing:**(NSTIFFCompression)*compression*
Returns YES if the image can be compressed using the specified type of compression.

– (void)**getCompression:**(NSTIFFCompression *)*compression*
factor:(float *)*factor* Returns, in its arguments, the compression type and compression factor.

– (void)**setCompression:**(NSTIFFCompression)*compression*
factor:(float)*factor* Sets the compression type and compression factor.

NSBox

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSBox.h

Class Description

An NSBox object is a simple NSView that can do two things: It can draw a border around itself and it can title itself. You can use an NSBox to group, visually, some number of other NSViews. These other NSViews are added to the NSBox through the typical subview-adding methods, such as **addSubview:** and **replaceSubview:with:**.

An NSBox contains a *content area*, a rectangle set within the NSBox's frame in which the NSBox's subviews are displayed. The size and location of the content area depends on the NSBox's border type, title location, the size of the font used to draw the title, and an additional measure that you can set through the **setContentViewMargins:** method. When you create an NSBox, an instance of NSView is created and added (as a subview of the NSBox object) to fill the NSBox's content area. If you replace this *content view* with an NSView of your own, your NSView will be resized to fit the content area. Similarly, as you resize an NSBox its content view is automatically resized to fill the content area.

The NSViews that you add as subviews to an NSBox are actually added to the NSBox's content view—NSView's subview-adding methods are redefined by NSBox to ensure that a subview is correctly placed in the view hierarchy. However, you should note that the **subviews** method *isn't* redefined: It returns an NSArray containing a single object, the NSBox's content view.

Getting and Modifying the Border and Title

– (NSRect)**borderRect**	Returns the rectangle in which the border is drawn.
– (NSBorderType)**borderType**	Returns the box's border type.
– (void)**setBorderType:**(NSBorderType)*aType*	Sets the box's border to *aType*.
– (void)**setTitle:**(NSString *)*aString*	Sets the box's title to *aString*.
– (void)**setTitleFont:**(NSFont *)*fontObj*	Sets the NSFont of the title to *fontObj*.
– (void)**setTitlePosition:**(NSTitlePosition)*aPosition*	Sets the position of the title to *aPosition*.
– (NSString *)**title**	Returns the title of the box.
– (id)**titleCell**	Returns the Cell used to draw the title.
– (NSFont *)**titleFont**	Returns the NSFont used to draw the title.

– (NSTitlePosition)**titlePosition** Returns the position of the title.

– (NSRect)**titleRect** Returns the rectangle in which the title is drawn.

Setting and Placing the Content View

– (id)**contentView** Returns the content view.

– (NSSize)**contentViewMargins** Gets the distances between the border and the content view.

– (void)**setContentView:**(NSView *)*aView* Replaces the NSBox's content view with *aView*.

– (void)**setContentViewMargins:**(NSSize)*offsetSize* Sets the distances between the border and the content view to the horizontal and vertical amounts in *offsetSize*.

Resizing the Box

– (void)**setFrameFromContentFrame:**(NSRect)*contentFrame* Resizes the box to accommodate *contentFrame*.

– (void)**sizeToFit** Resizes the box to exactly enclose its subviews.

NSBrowser

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSBrowser.h

Class Description

NSBrowser provides a user interface for displaying and selecting items from a list of data, or from hierarchically organized lists of data such as directory paths. When working with a hierarchy of data, the levels are displayed in columns, which are numbered from left to right, beginning with 0. Each column consists of an NSScrollView containing an NSMatrix filled with NSBrowserCells. NSBrowser relies on a delegate to provide the data in its NSBrowserCells. See the NSBrowserCell class description for more on its implementation.

Browser Selection

An entry in an NSBrowser's column can be either a branch node (such as a directory) or a leaf node (such as a file). When the user selects a single branch node entry in a column, the NSBrowser sends itself the **addColumn** message, which messages its delegate to load the next column. The user's selection can be represented as a character string; if the selection is hierarchical (for example, a filename within a directory), each component of the path to the selected node is separated by "/". To use some other character as the delimiter, invoke **setPathSeparator:**.

An NSBrowser can be set to allow selection of multiple entries in a column, or to limit selection to a single entry. When set for multiple selection, it can also be set to limit multiple selection to leaf nodes only, or to allow selection of both types of nodes together.

As a subclass of NSControl, NSBrowser has a target object and action message. Each time the user selects one or more entries in a column, the action message is sent to the target. NSBrowser also adds an action to be sent when the user double-clicks on an entry, which allows the user to select items without any action being taken, and then double-click to invoke some useful action such as opening a file.

User Interface Features

The user interface features of an NSBrowser can be changed in a number of ways. The NSBrowser may or may not have a horizontal scroller. (The NSBrowser's columns, by contrast, always have vertical scrollers—although a scroller's buttons and knob might be invisible if the column doesn't contain many entries.) You generally shouldn't create an NSBrowser without a horizontal scroller; if you do, you must make sure the bounds rectangle of the NSBrowser is wide enough that all the columns can be displayed. An NSBrowser's columns may be bordered and titled, bordered and untitled, or unbordered and untitled. A column's title may be taken from the selected entry in the column to its left, or may be provided explicitly by the NSBrowser or its delegate.

NSBrowser's Delegate

NSBrowser requires a delegate to provide it with data to display. The delegate is responsible for providing the data and for setting each item as a branch or leaf node, enabled or disabled. It can also receive notification of events like scrolling and requests for validation of columns that may have changed.

You can implement one of two delegate types: active or passive. An active delegate creates a column's rows (that is, the NSBrowserCells) itself, while a passive one leaves that job to the NSBrowser. Normally, passive delegates are preferable, because they're easier to implement. An active delegate must implement **browser:createRowsForColumn:inMatrix:** to create the rows of the specified column. A passive delegate, on the other hand, must implement **browser:numberOfRowsInColumn:** to let the NSBrowser know how many rows to create. These two methods are mutually exclusive; you can implement one or the other, but not both. (The NSBrowser ascertains what type of delegate it has by which method the delegate responds to.)

Both types of delegate implement **browser:willDisplayCell:atRow:column:** to set up state (such as the cell's string value and whether the cell is a leaf or a branch) before an individual cell is displayed. (This delegate method doesn't need to invoke NSBrowserCell's **setLoaded:** method, because the NSBrowser can determine that state by itself.) An active delegate can instead set all the cells' state at the time the cells are created, in which case it doesn't need to implement **browser:willDisplayCell:atRow:column:**. However, a passive delegate must always implement this method.

Setting the Delegate

– (id)**delegate**

Returns the NSBrowser's delegate.

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

Sets the NSBrowser's delegate to *anObject*. Raises NSBrowserIllegalDelegateException if the delegate specified by *anObject* doesn't respond to **browser:willDisplayCell:atRow:column:** and either of the methods **browser:numberOfRowsInColumn:** or **browser:createRowsForColumn:inMatrix:**

Target and Action

– (SEL)**doubleAction**

Returns the NSBrowser's double-click action method.

– (BOOL)**sendAction**

Sends the action message to the target. Returns YES upon success, NO if no responder for the message could be found.

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

Sets the NSBrowser's double-click action to *aSelector.*

Setting Component Classes

+ (Class)**cellClass**

Returns the NSBrowserCell class (regardless of whether a **setCellClass:** message has been sent to a particular instance).

– (id)**cellPrototype** Returns the NSBrowser's prototype NSCell.

– (Class)**matrixClass** Returns the class of NSMatrix used in the NSBrowser's columns.

– (void)**setCellClass:**(Class)*classId* Sets the class of NSCell used in the columns of the NSBrowser.

– (void)**setCellPrototype:**(NSCell *)*aCell* Sets the NSCell instance copied to display items in the columns of NSBrowser.

– (void)**setMatrixClass:**(Class)*classId* Sets the class of NSMatrix used in the NSBrowser's columns.

Setting NSBrowser Behavior

– (BOOL)**reusesColumns** Returns YES if NSMatrix objects aren't freed when their columns are unloaded.

– (void)**setReusesColumns:**(BOOL)*flag* If *flag* is YES, prevents NSMatrix objects from being freed when their columns are unloaded, so they can be reused.

– (void)**setTakesTitleFromPreviousColumn:**(BOOL)*flag* Sets whether the title of a column is set to the string value of the selected NSCell in the previous column.

– (BOOL)**takesTitleFromPreviousColumn** Returns YES if the title of a column is set to the string value of the selected NSCell in the previous column.

Allowing Different Types of Selection

– (BOOL)**allowsBranchSelection** Returns whether the user can select branch items when multiple selection is enabled.

– (BOOL)**allowsEmptySelection** Returns whether there can be nothing selected.

– (BOOL)**allowsMultipleSelection** Returns whether the user can select multiple items.

– (void)**setAllowsBranchSelection:**(BOOL)*flag* Sets whether the user can select branch items when multiple selection is enabled.

– (void)**setAllowsEmptySelection:**(BOOL)*flag* Sets whether there can be nothing selected.

– (void)**setAllowsMultipleSelection:**(BOOL)*flag* Sets whether the user can select multiple items.

Setting Arrow Key Behavior

– (BOOL)**acceptsArrowKeys** Returns YES if the arrow keys are enabled.

– (BOOL)**sendsActionOnArrowKeys** Returns NO if pressing an arrow key only scrolls the browser, YES if it also sends the action message specified by **setAction:**.

– (void)**setAcceptsArrowKeys:**(BOOL)*flag* Enables or disables the arrow keys.

– (void)**setSendsActionOnArrowKeys:**(BOOL)*flag* Sets whether pressing an arrow key will cause the action message to be sent (in addition to causing scrolling).

Showing a Horizontal Scroller

– (void)**setHasHorizontalScroller:**(BOOL)*flag* Sets whether an NSScroller is used to scroll horizontally.

– (BOOL)**hasHorizontalScroller** Returns whether an NSScroller is used to scroll horizontally.

Setting the NSBrowser's Appearance

– (int)**maxVisibleColumns** Returns the maximum number of visible columns.

– (int)**minColumnWidth** Returns the minimum column width.

– (BOOL)**separatesColumns** Returns whether columns are separated by bezeled borders.

– (void)**setMaxVisibleColumns:**(int)*columnCount* Sets the maximum number of columns displayed.

– (void)**setMinColumnWidth:**(int)*columnWidth* Sets the minimum column width.

– (void)**setSeparatesColumns:**(BOOL)*flag* Sets whether to separate columns with bezeled borders.

Manipulating Columns

– (void)**addColumn** Adds a column to the right of the last column.

– (int)**columnOfMatrix:**(NSMatrix *)*matrix* Returns the column number in which *matrix* is located.

– (void)**displayAllColumns** Updates the NSBrowser to display all loaded columns.

– (void)**displayColumn:**(int)*column* Updates the NSBrowser to display the column with the given index.

– (int)**firstVisibleColumn** Returns the index of the first visible column.

– (BOOL)**isLoaded** Returns whether column zero is loaded.

– (int)**lastColumn** Returns the index of the last column loaded.

– (int)**lastVisibleColumn** Returns the index of the last visible column.

– (void)**loadColumnZero** Loads column zero; unloads previously loaded columns.

– (int)**numberOfVisibleColumns**	Returns the number of columns visible.
– (void)**reloadColumn:**(int)*column*	Reloads *column* if it is loaded; sets it as the last column.
– (void)**selectAll:**(id)*sender*	Selects all NSCells in the last column of the NSBrowser.
– (int)**selectedColumn**	Returns the index of the last column with a selected item.
– (void)**setLastColumn:**(int)*column*	Sets the last column to *column*.
– (void)**validateVisibleColumns**	Invokes delegate method **browser:isColumnValid:** for visible columns.

Manipulating Column Titles

– (void)**drawTitle:**(NSString *)*title* **inRect:**(NSRect)*aRect* **ofColumn:**(int)*column*	Draws the title for the column at index *column*.
– (BOOL)**isTitled**	Returns whether columns display titles.
– (void)**setTitled:**(BOOL)*flag*	Sets whether columns display titles.
– (void)**setTitle:**(NSString *)*aString* **ofColumn:**(int)*column*	Sets the title of the column at index *column* to *aString*.
– (NSRect)**titleFrameOfColumn:**(int)*column*	Returns the bounds of the title frame for the column at index *column*.
– (float)**titleHeight**	Returns the height of column titles.
– (NSString *)**titleOfColumn:**(int)*column*	Returns the title displayed for the column at index *column*.

Scrolling an NSBrowser

– (void)**scrollColumnsLeftBy:**(int)*shiftAmount*	Scrolls columns left by *shiftAmount* columns.
– (void)**scrollColumnsRightBy:**(int)*shiftAmount*	Scrolls columns right by *shiftAmount* columns.
– (void)**scrollColumnToVisible:**(int)*column*	Scrolls to make the column at index *column* visible.
– (void)**scrollViaScroller:**(NSScroller *)*sender*	Scrolls columns left or right based on an NSScroller.
– (void)**updateScroller**	Updates the horizontal scroller to reflect column positions.

Event Handling

– (void)**doClick:**(id)*sender*	Responds to mouse clicks in a column of the NSBrowser.
– (void)**doDoubleClick:**(id)*sender*	Responds to double-clicks in a column of the NSBrowser.

Getting Matrices and Cells

– (id)**loadedCellAtRow:**(int)*row* **column:**(int)*column*	Loads if necessary and returns the NSCell at *row* in *column*.
– (NSMatrix *)**matrixInColumn:**(int)*column*	Returns the matrix located in *column*.
– (id)**selectedCell**	Returns the last (rightmost and lowest) selected NSCell.
– (id)**selectedCellInColumn:**(int)*column*	Returns the last (lowest) NSCell that's selected in *column*.
– (NSArray *)**selectedCells**	Returns all the rightmost selected NSCells.

Getting Column Frames

– (NSRect)**frameOfColumn:**(int)*column*	Returns the rectangle containing the column at index *column*.
– (NSRect)**frameOfInsideOfColumn:**(int)*column*	Returns the rectangle containing the column at index *column*, not including borders.

Manipulating Paths

– (NSString *)**path**	Returns the browser's current path.
– (NSString *)**pathSeparator**	Returns the path separator. The default is "/".
– (NSString *)**pathToColumn:**(int)*column*	Returns a string representing the path from the first column to the column at index *column*.
– (BOOL)**setPath:**(NSString *)*path*	Parses *path* and selects corresponding items in columns.
– (void)**setPathSeparator:**(NSString *)*aString*	Sets the path separator to *aString*.

Arranging an NSBrowser's Components

– (void)**tile**	Adjusts the various subviews of NSBrowser—scrollers, columns, titles, and so on—without redrawing. Your code shouldn't send this message. It's invoked any time the appearance of the NSBrowser changes.

Methods Implemented by the Delegate

– (void)**browser:**(NSBrowser *)*sender* **createRowsForColumn:**(int)*column* **inMatrix:**(NSMatrix *)*matrix*	Creates a row in *matrix* for each row of data to be displayed in *column* of the browser. Either this method or **browser:numberOfRowsInColumn:** must be implemented, but not both (or an NSBrowserIllegalDelegateException will be raised).

– (BOOL)**browser:**(NSBrowser *)*sender* **isColumnValid:**(int)*column*	Returns whether the contents of the specified column are valid.
– (int)**browser:**(NSBrowser *)*sender* **numberOfRowsInColumn:**(int)*column*	Returns the number of rows of data in the column at index *column*. Either this method or **browser:createRowsForColumn:inMatrix:** must be implemented, but not both.
– (BOOL)**browser:**(NSBrowser *)*sender* **selectCell:**(NSString *)*title* **inColumn:**(int)*column*	Asks the delegate to select the NSCell with title *title* in the column at index *column*.
– (NSString *)**browser:**(NSBrowser *)*sender* **titleOfColumn:**(int)*column*	Queries the delegate for the title to display above the column at index *column*.
– (void)**browser:**(NSBrowser *)*sender* **willDisplayCell:**(id)*cell* **atRow:**(int)*row* **column:**(int)*column*	Notifies the delegate when the NSBrowser will display the specified cell. The delegate should set any state necessary for correct display of the cell.
– (void)**browserDidScroll:**(NSBrowser *)*sender*	Notifies the delegate when the NSBrowser has scrolled.
– (void)**browserWillScroll:**(NSBrowser *)*sender*	Notifies the delegate when the NSBrowser will scroll.

NSBrowserCell

Inherits From: NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSBrowserCell.h

Class Description

NSBrowserCell is the subclass of NSCell used by default to display data in the columns of an NSBrowser. (Each column contains an NSMatrix filled with NSBrowserCells.) Many of NSBrowserCell's methods are designed to interact with NSBrowser and NSBrowser's delegate. The delegate implements methods for loading the NSCells in NSBrowser by setting their values and status. If your code needs access to a specific NSBrowserCell, you can use the NSBrowser method **loadedCellAtRow:column:**.

You may find it useful to create a subclass of NSBrowserCell to alter its behavior and to enable it to work with and display the type of data you wish to represent. Use NSBrowser's **setCellClass:** or **setCellPrototype:** methods to have it use your subclass.

See the NSBrowser class specification for more details. In particular, the class description and the "Methods Implemented by the Delegate" section describe how the NSBrowser's delegate interacts with both NSBrowser and NSBrowserCells.

Accessing Graphic Attributes

+ (NSImage *)**branchImage**	Returns the default NSImage for branch NSBrowserCells.
+ (NSImage *)**highlightedBranchImage**	Returns the default NSImage for branch NSBrowserCells that are highlighted.
– (NSImage *)**alternateImage**	Returns this NSBrowserCell's image for the highlighted state.
– (void)**setAlternateImage:**(NSImage *)*anImage*	Sets this NSBrowserCell's image for the highlighted state.

Placing in the Browser Hierarchy

– (BOOL)**isLeaf**	Returns whether the NSBrowserCell is a leaf or a branch.
– (void)**setLeaf:**(BOOL)*flag*	Sets whether the NSBrowserCell is a leaf or a branch.

Determining Loaded Status

– (BOOL)**isLoaded**

Returns YES if all the NSBrowserCell's state has been set and the cell is ready to display.

– (void)**setLoaded:**(BOOL)*flag*

Sets whether all the NSBrowserCell's state has been set and the cell is ready to display.

Setting State

– (void)**reset**

Unhighlights the NSBrowserCell and sets its state to 0.

– (void)**set**

Highlights the NSBrowserCell and sets its state to 1.

NSBundle Additions

Inherits From: NSObject

Declared In: AppKit/NSImage.h
AppKit/NSNibLoading.h

Class Description

The Application Kit adds these methods to the Foundation Kit's NSBundle class. These methods become part of the class for all applications that use the Application Kit, but not for applications that don't.

Getting the Location of Images in the File System

– (NSString *)**pathForImageResource:**(NSString *)*name*

Returns the absolute pathname of the file containing the specified image resource. (The *name* of the resource is simply the filename without the path of its bundle directory; the filename extension need not be included.)

Loading an Interface Builder File

+ (BOOL)**loadNibFile:**(NSString *)*fileName* **externalNameTable:**(NSDictionary *)*context* **withZone:**(NSZone *)*zone*

Unarchives the contents of the nib file whose absolute path is *fileName*. Objects from the nib file are allocated in the specified zone of memory. The *context* argument is a name table—a dictionary whose keys are names like "NSOwner" and whose values are existing objects that can be referenced by the newly unarchived objects. Returns YES upon success. (A nib file is a object archive whose file format is currently implementation specific. A public specification of this file format will be available at a later date.)

+ (BOOL)**loadNibNamed:**(NSString *)*aNibName* **owner:**(id)*owner*

Similar to **loadNibFile:externalNameTable:withZone:**, but the name table's only element is the specified owner (stored with the key "NSOwner"). Objects from the nib file are allocated in *owner*'s zone. If there's a bundle for *owner*'s class, this method looks in that bundle for the nib file named *aNibName* (this argument need not include the ".nib" extension); otherwise, it looks in the main bundle. (A nib file is a object archive whose file format is currently implementation specific. A public specification of this file format will be available at a later date.)

NSButton

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSButton.h

Class Description

NSButton is a subclass of NSControl that intercepts mouse-down events and sends an action message to a target object when it's clicked or pressed. By virtue of its NSButtonCell, NSButton is a two-state NSControl—it's either "off" or "on"—and it displays its state depending on the configuration of the NSButtonCell. NSButton acquires other attributes of NSButtonCell. The state is used as the value, so NSControl methods like **setIntValue:** actually set the state (the methods **setState:** and **state** are provided as a more conceptually accurate way of setting and getting the state). The NSButton can send its action continuously and display highlighting in several different ways. What's more, an NSButton can have a key equivalent that's eligible for triggering whenever the NSButton's NSPanel or NSWindow is the key window.

NSButton and NSMatrix both provide a control view, which is needed to display an NSButtonCell object. However, while NSMatrix requires you to access the NSButtonCells directly, most of NSButton's methods are "covers" for identically declared methods in NSButtonCell. (In other words, the implementation of the NSButton method invokes the corresponding NSButtonCell method for you, allowing you to be unconcerned with the NSButtonCell's existence.) The only NSButtonCell methods that don't have covers relate to the font used to display the key equivalent, and to specific methods for highlighting or showing the NSButton's state (these last are usually set together with NSButton's **setType:** method).

Creating a Subclass of NSButton

Override the designated initializer (NSView's **initWithFrame:** method) if you create a subclass of NSButton that performs its own initialization. If you want to use a custom NSButtonCell subclass with your subclass of NSButton, you have to override the **setCellClass:** method, as described in "Creating New NSControls" in the NSControl class specification.

See the NSButtonCell class specification for more on NSButton's behavior.

Initializing the NSButton Factory

+ (Class)**cellClass**	Returns the subclass of NSButtonCell used by NSButton**.**
+ (void)**setCellClass:**(Class)*classId*	Sets the subclass of NSButtonCell used by NSButton.

Setting the Button Type

– (void)**setType:**(int)*aType* — Sets how the NSButton highlights and shows its state.

Setting the State

– (void)**setState:**(int)*value* — Sets the NSButton's state to *value* (0 or 1).

– (int)**state** — Returns the NSButton's current state (0 or 1).

Setting the Repeat Interval

– (void)**getPeriodicDelay:**(float *)*delay*
interval:(float *)*interval* — Gets repeat parameters for continuous buttons.

– (void)**setPeriodicDelay:**(float)*delay*
interval:(float)*interval* — Sets repeat parameters for continuous buttons.

Setting the Titles

– (NSString *)**alternateTitle** — Returns the button's alternate title.

– (void)**setAlternateTitle:**(NSString *)*aString* — Makes *aString* the button's alternate title.

– (void)**setTitle:**(NSString *)*aString* — Makes *aString* the button's title.

– (NSString *)**title** — Returns the button's title.

Setting the Images

– (NSImage *)**alternateImage** — Returns the button's alternate image.

– (NSImage *)**image** — Returns the button's image.

– (NSCellImagePosition)**imagePosition** — Returns the position of the button's image.

– (void)**setAlternateImage:**(NSImage *)*anImage* — Makes *anImage* the alternate image.

– (void)**setImage:**(NSImage *)*anImage* — Makes *anImage* the button's icon.

– (void)**setImagePosition:**(NSCellImagePosition)*aPosition* — Sets the position of the button's image to *aPosition*.

Modifying Graphic Attributes

– (BOOL)**isBordered**	Returns whether the button has a bezeled border.
– (BOOL)**isTransparent**	Returns whether the button is transparent.
– (void)**setBordered:**(BOOL)*flag*	Sets whether the button has a bezeled border.
– (void)**setTransparent:**(BOOL)*flag*	Sets whether the button is transparent.

Displaying

– (void)**highlight:**(BOOL)*flag*	Highlights (or unhighlights) the button according to *flag*.

Setting the Key Equivalent

– (NSString *)**keyEquivalent**	Returns the button's key equivalent.
– (unsigned int)**keyEquivalentModifierMask**	Returns the mask indicating the possible modifier keys for button's key equivalent.
– (void)**setKeyEquivalent:**(NSString *)*aKeyEquivalent*	Makes *aKeyEquivalent* the button's key equivalent.
– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*	Sets the mask that determines the possible modifier keys for button's key equivalent.

Handling Events and Action Messages

– (void)**performClick:**(id)*sender*	Simulates the user's clicking the button.
– (BOOL)**performKeyEquivalent:**(NSEvent *)*anEvent*	Simulates a mouse click, if the key in *anEvent* is right.

NSButtonCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSButtonCell.h

Class Description

NSButtonCell is a subclass of NSActionCell used to implement the user interfaces of push buttons, switches, and radio buttons. It can also be used for any other region of a view that's designed to send a message to a target when clicked. The NSButton subclass of NSControl uses a single NSButtonCell. To create groups of switches or radio buttons, use an NSMatrix holding a set of NSButtonCells.

An NSButtonCell is a two-state cell; it's either "off" or "on," and can be configured to display the two states differently, with a separate title and/or image for either state. The two states are more often referred to as "normal" and "alternate." An NSButtonCell's state is also used as its value, so NSCell methods that set the value (**setIntValue:** and so on) actually set the NSButtonCell's state to "on" if the value provided is non-zero (or non-null for strings), and to "off" if the value is zero or null. Similarly, methods that retrieve the value return 1 for the "on" or alternate state (an empty string in the case of **stringValue**), or 0 or NULL for the "off" or normal state. You can also use NSCell's **setState:** and **state** methods to set or retrieve the state directly. After changing the state, send a **display** message to show the NSButtonCell's new appearance. (NSButton does this automatically.)

An NSButtonCell sends its action message to its target once if its view is clicked and it gets the mouse-down event, but can also send the action message continuously as long as the mouse is held down with the cursor inside the NSButtonCell. The NSButtonCell can show that it's being pressed by highlighting in several ways—for example, a bordered NSButtonCell can appear pushed into the screen, or the image or title can change to an alternate form while the NSButtonCell is pressed.

An NSButtonCell can also have a key equivalent (like a menu item). If the NSButtonCell is displayed in the key window, the NSButtonCell gets the first chance to receive events related to key equivalents. This feature is used quite often in modal panels that have an "OK" button containing the image that represents the Return key. Usually an NSButtonCell displays a key equivalent as its image; if you ever set an image for the NSButtonCell, the key equivalent remains, but doesn't get displayed.

For more information on NSButtonCell's behavior, see the NSButton and NSMatrix class specifications.

Exceptions

In its implementation of the **compare:** method (declared in NSCell), NSButtonCell raises NSBadComparisonException if the *otherCell* argument is not of the NSButtonCell class.

Setting the Titles

– (NSString *)**alternateTitle** — Returns the NSButtonCell's alternate title (used while the button is in the highlighted state).

– (void)**setAlternateTitle:**(NSString *)*aString* — Makes a copy of *aString* and uses it as the NSButtonCell's alternate title.

– (void)**setFont:**(NSFont *)*fontObject* — Sets the NSFont used to draw the title.

– (void)**setTitle:**(NSString *)*aString* — Makes a copy of *aString* and uses it as the NSButtonCell's title.

– (NSString *)**title** — Returns the NSButtonCell's title.

Setting the Images

– (NSImage *)**alternateImage** — Returns the NSButtonCell's alternate image (used while the button is in the highlighted state).

– (NSCellImagePosition)**imagePosition** — Returns the position of the NSButtonCell's image.

– (void)**setAlternateImage:**(NSImage *)*anImage* — Makes *anImage* the alternate image.

– (void)**setImagePosition:**(NSCellImagePosition)*aPosition* — Sets the position of the NSButtonCell's image in relation to its title.

Setting the Repeat Interval

– (void)**getPeriodicDelay:**(float *)*delay* **interval:**(float *)*interval* — Gets repeat parameters for continuous NSButtonCells.

– (void)**setPeriodicDelay:**(float)*delay* **interval:**(float)*interval* — Sets repeat parameters for continuous NSButtonCells.

Setting the Key Equivalent

– (NSString *)**keyEquivalent** — Returns the NSButtonCell's key equivalent.

– (NSFont *)**keyEquivalentFont** — Returns the NSFont used to draw the key equivalent.

– (unsigned int)**keyEquivalentModifierMask** — Returns the mask indicating the possible modifier keys for NSButtonCell's key equivalent.

– (void)**setKeyEquivalent:**(NSString *)*aKeyEquivalent* — Sets the NSButtonCell's key equivalent.

– (void)**setKeyEquivalentModifierMask:**(unsigned int)*mask*
Sets the mask that determines the possible modifier keys for NSButtonCell's key equivalent.

– (void)**setKeyEquivalentFont:**(NSFont *)*fontObj* Sets the NSFont used to draw the key equivalent.

– (void)**setKeyEquivalentFont:**(NSString *)*fontName*
size:(float)*fontSize* Sets the NSFont and size used to draw the key equivalent.

Modifying Graphic Attributes

– (BOOL)**isOpaque** Returns whether receiver is opaque.

– (BOOL)**isTransparent** Returns whether the NSButtonCell is transparent.

– (void)**setTransparent:**(BOOL)*flag* Sets whether the NSButtonCell is transparent.

Modifying Graphic Attributes

– (int)**highlightsBy** Returns how the NSButtonCell highlights when pressed.

– (void)**setHighlightsBy:**(int)*aType* Sets how the NSButtonCell highlights when pressed.

– (void)**setShowsStateBy:**(int)*aType* Sets how the NSButtonCell shows its alternate (pressed) state.

– (void)**setType:**(NSButtonType)*aType* Sets the NSButtonCell's display behavior.

– (int)**showsStateBy** Returns how NSButtonCell shows its alternate (pressed) state.

Simulating a Click

– (void)**performClick:**(id)*sender* Simulates a user's mouse click on the NSButtonCell.

NSCachedImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding, NSCopying (NSImageRep)
NSObject (NSObject)

Declared In: AppKit/NSCachedImageRep.h

Class Description

NSCachedImageRep, a subclass of NSImageRep, defines an object that stores its source data as a rendered image in a window, typically a window that stays off-screen. The only data that's available for reproducing the image is the image itself. Thus an NSCachedImageRep differs from the other kinds of NSImageReps defined in the Application Kit, all of which can reproduce an image from the information originally used to draw it. Instances of this class are generally used indirectly, through an NSImage object.

Initializing an NSCachedImageRep

– (id)**initWithSize:**(NSSize)*aSize*
depth:(NSWindowDepth)*aDepth*
separate:(BOOL)*separate*
alpha:(BOOL)*alpha*

Initializes a new NSCachedImageRep for an image of the specified size and depth. The *separate* argument specifies whether the image will get its own unique cache, instead of possibly sharing one with other images. For best performance (although it's not essential), the *alpha* argument should be set according to whether the image will have a channel for transparency information.

– (id)**initWithWindow:**(NSWindow *)*aWindow*
rect:(NSRect)*aRect*

Initializes the new NSCachedImageRep for an image to be drawn in the rectangle *aRect* of the specified window. This method retains *aWindow.*

Getting the Representation

– (NSRect)**rect**

Returns the rectangle where the image is cached.

– (NSWindow *)**window**

Returns the NSWindow where the image is cached.

NSCell

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSCell.h

Class Description

The NSCell class provides a mechanism for displaying text or images in an NSView without the overhead of a full NSView subclass. In particular, it provides much of the functionality of the NSText class by providing access to a shared NSText object used by all instances of NSCell in an application. NSCells are also extremely useful for placing titles or images at various locations in a custom subclass of NSView.

NSCell is used heavily by most of the NSControl classes to implement their internal workings. For example, NSSlider uses an NSSliderCell, NSTextField uses an NSTextFieldCell, and NSBrowser uses an NSBrowserCell. Sending a message to the NSControl is often simpler than dealing directly with the corresponding NSCell. For instance, NSControls typically invoke **updateCell:** (causing the cell to be displayed) after changing a cell attribute; whereas if you directly call the corresponding method of the NSCell, the NSCell might not automatically display itself again.

Some subclasses of NSControl (notably NSMatrix) allow multiple NSCells to be grouped and to act together in some cooperative manner. Thus, with an NSMatrix, a group of radio buttons can be implemented without needing an NSView for each button (and without needing an NSText object for the text on each button).

The NSCell class provides primitives for displaying text or an image, editing text, formatting floating-point numbers, maintaining state, highlighting, and tracking the mouse. NSCell's method **trackMouse:inRect:ofView:untilMouseUp:** supports the target object and action method used to implement controls. However, NSCell implements target/action features abstractly, deferring the details of implementation to subclasses of NSActionCell.

The **initImageCell:** method is the designated initializer for NSCells that display images. The **initTextCell:** method is the designated initializer for NSCells that display text. Override one or both of these methods if you implement a subclass of NSCell that performs its own initialization. If you need to use target and action behavior, you may prefer to subclass NSActionCell, which provides the default implementation of this behavior.

For more information on how NSCell is used, see the NSControl class specification.

Initializing an NSCell

– (id)**initImageCell:**(NSImage *)*anImage*	Initializes a new NSCell with the NSImage *anImage.*
– (id)**initTextCell:**(NSString *)*aString*	Initializes a new NSCell with title *aString*.

Determining Component Sizes

– (void)**calcDrawInfo:**(NSRect)*aRect*	Implemented by subclasses to recalculate drawing sizes.
– (NSSize)**cellSize**	Returns the minimum size needed to display the NSCell.
– (NSSize)**cellSizeForBounds:**(NSRect)*aRect*	Returns the minimum size needed to display the NSCell.
– (NSRect)**drawingRectForBounds:**(NSRect)*theRect*	Returns the rectangle the NSCell draws in.
– (NSRect)**imageRectForBounds:**(NSRect)*theRect*	Returns the rectangle that the cell's image is drawn in.
– (NSRect)**titleRectForBounds:**(NSRect)*theRect*	Returns the rectangle that the cell's title is drawn in.

Setting the NSCell's Type

– (void)**setType:**(NSCellType)*aType*	Sets the NSCell's type to *aType*.
– (NSCellType)**type**	Returns the NSCell's type.

Setting the NSCell's State

– (void)**setState:**(int)*value*	Sets the state of the NSCell to *value* (0 or 1).
– (int)**state**	Returns the state of the NSCell (0 or 1).

Enabling and Disabling the NSCell

– (BOOL)**isEnabled**	Returns whether the NSCell reacts to mouse events.
– (void)**setEnabled:**(BOOL)*flag*	Sets whether the NSCell reacts to mouse events.

Setting the Image

– (NSImage *)**image**	Returns the NSCell's image.
– (void)**setImage:**(NSImage *)*anImage*	Makes *anImage* the NSCell's image.

Setting the NSCell's Value

– (double)**doubleValue**	Returns the NSCell's value as a **double**.
– (float)**floatValue**	Returns the NSCell's value as a **float**.
– (int)**intValue**	Returns the NSCell's value as an **int**.
– (NSString *)**stringValue**	Returns the NSCell's value as a string.

– (void)**setDoubleValue:**(double)*aDouble* Sets the NSCell's value to *aDouble*.

– (void)**setFloatValue:**(float)*aFloat* Sets the NSCell's value to *aFloat*.

– (void)**setIntValue:**(int)*anInt* Sets the NSCell's value to *anInt*.

– (void)**setStringValue:**(NSString *)*aString* Sets the NSCell's value to a copy of *aString*.

Interacting with Other NSCells

– (void)**takeDoubleValueFrom:**(id)*sender* Sets the NSCell's value to *sender*'s double floating-point value.

– (void)**takeFloatValueFrom:**(id)*sender* Sets the NSCell's value to *sender*'s floating-point value.

– (void)**takeIntValueFrom:**(id)*sender* Sets the NSCell's value to *sender*'s integer value.

– (void)**takeStringValueFrom:**(id)*sender* Sets the NSCell's value to *sender*'s string value.

Modifying Text Attributes

– (NSTextAlignment)**alignment** Returns the alignment of text in the NSCell.

– (NSFont *)**font** Returns the Font used to display text in the NSCell.

– (BOOL)**isEditable** Returns whether the NSCell's text is editable.

– (BOOL)**isSelectable** Returns whether the NSCell's text is selectable.

– (BOOL)**isScrollable** Returns whether the NSCell scrolls to follow typing.

– (void)**setAlignment:**(NSTextAlignment)*mode* Sets the alignment of text in the NSCell to *mode*.

– (void)**setEditable:**(BOOL)*flag* Sets whether the NSCell's text is editable.

– (void)**setFont:**(NSFont *)*fontObject* Sets the Font used to display text in the NSCell to *fontObject*.

– (void)**setSelectable:**(BOOL)*flag* Sets whether the NSCell's text is selectable.

– (void)**setScrollable:**(BOOL)*flag* Sets whether the NSCell scrolls to follow typing.

– (NSText *)**setUpFieldEditorAttributes:**(NSText *)*textObject*
Sets NSText parameters for the field editor. (See the documentation for NSText.)

– (void)**setWraps:**(BOOL)*flag* Sets whether the NSCell's text is word-wrapped.

– (BOOL)**wraps** Returns whether the NSCell's text is word-wrapped.

Editing Text

– (void)**editWithFrame:**(NSRect)*aRect* **inView:**(NSView *)*controlView* **editor:**(NSText *)*textObject* **delegate:**(id)*anObject* **event:**(NSEvent *)*theEvent*	Allows text editing in response to a mouse-down event.
– (void)**endEditing:**(NSText *)*textObject*	Ends any text editing occurring in the NSCell.
– (void)**selectWithFrame:**(NSRect)*aRect* **inView:**(NSView *)*controlView* **editor:**(NSText *)*textObject* **delegate:**(id)*anObject* **start:**(int)*selStart* **length:**(int)*selLength*	Allows text selection in response to a mouse-down event.

Validating Input

– (int)**entryType**	Returns the type of data the user can type into the NSCell.
– (BOOL)**isEntryAcceptable:**(NSString *)*aString*	Returns whether *aString* is acceptable for the entry type.
– (void)**setEntryType:**(int)*aType*	Sets the type of data the user can type into the NSCell.

Formatting Data

– (void)**setFloatingPointFormat:**(BOOL)*autoRange* **left:**(unsigned int)*leftDigits* **right:**(unsigned int)*rightDigits*	Sets the display format for floating-point values.

Modifying Graphic Attributes

– (BOOL)**isBezeled**	Returns whether the NSCell has a bezeled border.
– (BOOL)**isBordered**	Returns whether NSCell has a plain border.
– (BOOL)**isOpaque**	Returns whether the NSCell is opaque.
– (void)**setBezeled:**(BOOL)*flag*	Sets whether the NSCell has a bezeled border.
– (void)**setBordered:**(BOOL)*flag*	Sets whether the NSCell has a plain border.

Setting Parameters

– (int)**cellAttribute:**(NSCellAttribute)*aParameter*
Returns various flag values.

– (void)**setCellAttribute:**(NSCellAttribute)*aParameter* **to:**(int)*value*
Sets various NSCell flags.

Displaying

– (NSView *)**controlView**
Implemented by subclasses to return the NSView last drawn in (normally an NSControl).

– (void)**drawInteriorWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*
Draws the area within the NSCell's border in *controlView.*

– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*
Draws the entire NSCell in *controlView.*

– (void)**highlight:**(BOOL)*lit* **withFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*
If *lit* is YES, highlights the NSCell in *controlView,* otherwise unhighlights.

– (BOOL)**isHighlighted**
Returns whether the NSCell is highlighted.

Target and Action

– (SEL)**action**
Implemented by subclasses to return the action method.

– (BOOL)**isContinuous**
Returns whether the NSCell continuously sends the action.

– (int)**sendActionOn:**(int)*mask*
Determines when the action is sent while tracking.

– (void)**setAction:**(SEL)*aSelector*
Implemented by subclasses to set the action method.

– (void)**setContinuous:**(BOOL)*flag*
Sets whether the NSCell continuously sends the action.

– (void)**setTarget:**(id)*anObject*
Implemented by subclasses to set the target object.

– (id)**target**
Implemented by subclasses to return the target object.

Assigning a Tag

– (void)**setTag:**(int)*anInt*
Implemented by subclasses to set an identifier tag.

– (int)**tag**
Implemented by subclasses to return the identifier tag.

Handling Keyboard Alternatives

– (NSString *)**keyEquivalent**
Implemented by subclasses to return a key equivalent.

Tracking the Mouse

+ (BOOL)**prefersTrackingUntilMouseUp**
Returns NO, so tracking stops when the mouse leaves the NSCell; subclasses may override.

– (BOOL)**continueTracking:**(NSPoint)*lastPoint*
at:(NSPoint)*currentPoint*
inView:(NSView *)*controlView*
Returns whether tracking should continue based on *lastPoint* and *currentPoint* within *controlView.*

– (int)**mouseDownFlags**
Returns the event flags set at the start of mouse tracking.

– (void)**getPeriodicDelay:**(float *)*delay*
interval:(float *)*interval*
Returns repeat values for continuous sending of the action.

– (BOOL)**startTrackingAt:**(NSPoint)*startPoint*
inView:(NSView *)*controlView*
Determines whether tracking should begin based on *startPoint* within *controlView.*

– (void)**stopTracking:**(NSPoint)*lastPoint*
at:(NSPoint)*stopPoint*
inView:(NSView *)*controlView*
mouseIsUp:(BOOL)*flag*
Allows the NSCell to update itself to end tracking, based on *lastPoint* and *stopPoint* within *controlView*; flag is YES if this method was invoked because the mouse went up.

– (BOOL)**trackMouse:**(NSEvent *)*theEvent*
inRect:(NSRect)*cellFrame*
ofView:(NSView *)*controlView*
untilMouseUp:(BOOL)*flag*
Tracks the mouse, returning YES if the mouse goes up while in *cellFrame*. This method is usually invoked by an NSControl's **mouseDown:** method, which passes the mouse-down event in *theEvent*. If *flag* is YES, the method keeps tracking until the mouse goes up; otherwise it tracks until the mouse leaves *cellFrame.*

Managing the Cursor

– (void)**resetCursorRect:**(NSRect)*cellFrame*
inView:(NSView *)*controlView*
Sets text NSCells to show the I-beam cursor.

Comparing to Another NSCell

– (NSComparisonResult)**compare:**(id)*otherCell*
Compares the string values of this cell and *otherCell* (which must be a kind of NSCell). Raises NSBadComparisonException if *otherCell* is not of the NSCell class.

Using the NSCell to Represent an Object

– (id)**representedObject**

Returns the object that the receiver represents, if any.

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

Creates an association between the receiver and *anObject*. *anObject* will be retained, released, archived, and unarchived whenever the receiver is. If another cell is already associated with *anObject*, that association is broken, and the receiver is associated with the object.

NSClipView

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSClipView.h

Class Description

An NSClipView object lets you scroll a document that may be larger than the NSClipView's frame rectangle, clipping the visible portion of the document to the frame. You don't normally use the NSClipView class directly; it's provided primarily as the scrolling machinery for the NSScrollView class. However, you might use the NSClipView class to implement a class similar to NSScrollView.

The document, which must be an NSView, is called the NSClipView's *document view.* An NSClipView's document view, which is set through the **setDocumentView:** method, is the NSClipView's only subview. You can set the cursor that's displayed when the mouse enters an NSClipView's frame (in other words, when it's poised over the document view) through the **setDocumentCursor:** method.

When the NSClipView is instructed to scroll its document view, it normally copies that portion of the document view that's visible both before and after the scrolling, so that this part won't need to be redrawn from scratch. However, you can turn off this behavior and force the entire visible area to be redrawn by sending the NSClipView a **setCopiesOnScroll:NO** message.

After scrolling, the NSClipView sends itself a **setNeedsDisplayInRect:** message to indicate that some part of the document view should be displayed again. The argument to this message is the freshly exposed area of the document view, unless the NSClipView received a **setCopiesOnScroll:NO** message, in which case the argument is the entire visible area.

The NSClipView sends its superview (usually an NSScrollView) a **reflectScrolledClipView:** message whenever the relationship between the NSClipView and the document view has changed. This allows the superview to update itself to reflect the change—for example, the NSScrollView class uses this method to change the position of its scrollers when the user causes the document to autoscroll.

Managing the Document V iew

– (NSRect)**documentRect**	Returns the document rectangle.
– (id)**documentView**	Returns the NSClipView's document view.
– (NSRect)**documentVisibleRect**	Gets the visible portion of the document view.
– (void)**setDocumentView:**(NSView *)*aView*	Makes *aView* the NSClipView's document view.

Setting the Cursor

–(NSCursor *)**documentCursor**	Returns the cursor for the document view.
– (void)**setDocumentCursor:**(NSCursor *)*anObject*	Sets the cursor for the document view.

Setting the Background Color

– (NSColor *)**backgroundColor**	Returns the NSClipView's background color.
– (void)**setBackgroundColor:**(NSColor *)*color*	Sets the NSClipView's background color.

Scrolling

– (BOOL)**autoscroll:**(NSEvent *)*theEvent*	Scrolls in response to mouse-dragged events.
– (NSPoint)**constrainScrollPoint:**(NSPoint)*newOrigin*	Prevents scrolling to an undesirable position.
– (BOOL)**copiesOnScroll**	Indicates whether the visible portions of the document view are copied when scrolling occurs. If not, the document view is responsible for redrawing the entire visible portion. The default is YES.
– (void)**scrollToPoint:**(NSPoint)*newOrigin*	Lowest-level unconstrained scrolling routine.
– (void)**setCopiesOnScroll:**(BOOL)*flag*	Sets how the visible areas are redrawn.

Responding to a Changed Frame

– (void)**viewFrameChanged:**(NSNotification *)*notification*	Notification that the document view's frame has changed.

NSCoder Additions

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSColor.h

Class Description

The Application Kit adds this method to the Foundation Kit's NSCoder class. This method becomes part of the class for all applications that use the Application Kit, but not for applications that don't.

Converting an Archived NXColor to an NSColor

– (NSColor *)**decodeNXColor**

Returns an autoreleased NSColor object equivalent to the archived NXColor structure. This method is needed to read colors from archives that were created by pre-OpenStep versions of NEXTSTEP.

NSColor

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSColor.h

An NSColor represents a color. The color can be a grayscale value and can include alpha (opacity) information. By sending a **set** message to an NSColor instance, you set the color for the current PostScript drawing context. This causes subsequently drawn graphics to have the color represented by the NSColor instance.

A color is defined in some particular *color space*. A color space consists of a set of dimensions—such as red, green, and blue in the case of RGB space. Each point in the space represents a unique color, and the point's location along each dimension is called a *component*. An individual color is usually specified by the numeric values of its components, which range from 0.0 to 1.0. For instance, a pure red is specified in RGB space by the component values 1.0, 0.0, and 0.0.

Some color spaces include an alpha component, which defines the color's opacity. An alpha value of 1.0 means completely opaque, and 0.0 means completely transparent. The alpha component is ignored when the color is used on a device that doesn't support alpha, such as a printer.

There are three kinds of color space in OpenStep:

- *Device-dependent*. This means that a given color might not look the same on different displays and printers.
- *Device-independent*, also known as *calibrated*. With this sort of color space, a given color should look the same on all devices.
- *Named*. The "named color space" has components that aren't numeric values, but simply names in various catalogs of colors. Named colors come with lookup tables that provide the ability to generate the correct color on a given device.

OpenStep includes six different color spaces, referred to by these enumeration constants:

NSDeviceCMYKColorSpace	Cyan, magenta, yellow, black, and alpha components
NSDeviceWhiteColorSpace	White and alpha components
NSDeviceRGBColorSpace	Red, green, blue, and alpha components Hue, saturation, brightness, and alpha components
NSCalibratedWhiteColorSpace	White and alpha components
NSCalibratedRGBColorSpace	Red, green, blue, and alpha components Hue, saturation, brightness, and alpha components
NSNamedColorSpace	Catalog name and color name components

(Color spaces whose names start with "NSDevice" are device-dependent; those with "NSCalibrated" are device-independent.)

There's usually no need to retrieve the individual components of a color, but when needed, you can either retrieve a set of components (using such methods as **getRed:green:blue:alpha:**) or an individual component (using such methods as **redComponent**). However, it's illegal to ask an NSColor for components that aren't defined for its color space. You can identify the color space by sending a **colorSpaceName** method to the NSColor. If you need to ask an NSColor for components that aren't in its color space (for instance, when you've gotten the color from the color panel), first convert the color to the appropriate color space using the **colorUsingColorSpaceName:** method. If the color is already in the specified color space, you get the same color back; otherwise you get a conversion that's usually lossy or that's correct only for the current device. You might also get back **nil** if the specified conversion can't be done.

Subclasses of NSColor need to implement the **colorSpaceName** and **set** methods, as well as the methods that return the components for that color space and the methods in the NSCoding protocol. Some other methods—such as **colorWithAlphaComponent:**, **isEqual:**, and **colorUsingColorSpaceName:device:**—may also be implemented if they make sense for the color space. Mutable subclasses (if any) should additionally implement **copyWithZone:** to provide a true copy.

Creating an NSColor from Component Values

+ (NSColor *)**colorWithCalibratedHue:**(float)*hue*
saturation:(float)*saturation*
brightness:(float)*brightness*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSCalibratedRGBColorSpace, whose opacity value is *alpha*, and whose components in HSB space would be *hue*, *saturation*, and *brightness*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

+ (NSColor *)**colorWithCalibratedRed:**(float)*red*
green:(float)*green*
blue:(float)*blue*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSCalibratedRGBColorSpace, whose opacity value is *alpha*, and whose RGB components are *red*, *green*, and *blue*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

+ (NSColor *)**colorWithCalibratedWhite:**(float)*white*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSCalibratedWhiteColorSpace, whose opacity value is *alpha*, and whose grayscale value is *white*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

+ (NSColor *)**colorWithCatalogName:**(NSString *)*listName*
colorName:(NSString *)*colorName*

Creates and returns a new NSColor whose color space is NSNamedColorSpace, by finding the color named *colorName* in the catalog named *listName.*

+ (NSColor *)**colorWithDeviceCyan:**(float)*cyan*
magenta:(float)*magenta*
yellow:(float)*yellow*
black:(float)*black*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSDeviceCMYKColorSpace, whose opacity value is *alpha*, and whose CMYK components are *cyan*, *magenta*, *yellow*, and *black*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

+ (NSColor *)**colorWithDeviceHue:**(float)*hue*
saturation:(float)*saturation*
brightness:(float)*brightness*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSDeviceRGBColorSpace, whose opacity value is *alpha*, and whose components in HSB space would be *hue*, *saturation*, and *brightness*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

+ (NSColor *)**colorWithDeviceRed:**(float)*red*
green:(float)*green*
blue:(float)*blue*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSDeviceRGBColorSpace, whose opacity value is *alpha*, and whose RGB components are *red*, *green*, and *blue*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

+ (NSColor *)**colorWithDeviceWhite:**(float)*white*
alpha:(float)*alpha*

Creates and returns a new NSColor whose color space is NSDeviceWhiteColorSpace, whose opacity value is *alpha*, and whose grayscale value is *white*. All values are legal, but values less than 0.0 are set to 0.0, and values greater than 1.0 are set to 1.0.

Creating an NSColor With Preset Components

+ (NSColor *)**blackColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 0.0 and whose alpha value is 1.0.

+ (NSColor *)**blueColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.0, 0.0, 1.0 and whose alpha value is 1.0.

+ (NSColor *)**brownColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.6, 0.4, 0.2 and whose alpha value is 1.0.

+ (NSColor *)**clearColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale and alpha values are both 0.0.

+ (NSColor *)**cyanColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.0, 1.0, 1.0 and whose alpha value is 1.0.

+ (NSColor *)**darkGrayColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 1/3 and whose alpha value is 1.0.

+ (NSColor *)**grayColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 0.5 and whose alpha value is 1.0.

+ (NSColor *)**greenColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.0, 1.0, 0.0 and whose alpha value is 1.0.

+ (NSColor *)**lightGrayColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale value is 2/3 and whose alpha value is 1.0.

+ (NSColor *)**magentaColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 0.0, 1.0 and whose alpha value is 1.0.

+ (NSColor *)**orangeColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 0.5, 0.0 and whose alpha value is 1.0.

+ (NSColor *)**purpleColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 0.5, 0.0, 0.5 and whose alpha value is 1.0.

+ (NSColor *)**redColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 0.0, 0.0 and whose alpha value is 1.0.

+ (NSColor *)**whiteColor**

Returns an NSColor in NSCalibratedWhiteColorSpace whose grayscale and alpha values are both 1.0.

+ (NSColor *)**yellowColor**

Returns an NSColor in NSCalibratedRGBColorSpace whose RGB value is 1.0, 1.0, 0.0 and whose alpha value is 1.0.

Ignoring Alpha Components

+ (BOOL)**ignoresAlpha**

Returns YES (the default) if the application hides the color panel's opacity slider and sets imported colors' alpha values to 1.0.

+ (void)**setIgnoresAlpha:**(BOOL)*flag*

If *flag* is YES, no opacity slider is displayed in the color panel, and colors dragged in or pasted have their alpha values set to 1.0.

Retrieving a Set of Components

– (void)**getCyan:**(float *)*cyan*
magenta:(float *)*magenta*
yellow:(float *)*yellow*
black:(float *)*black*
alpha:(float *)*alpha*

Returns the CMYK and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. It's an error if the receiver isn't a CMYK color.

– (void)**getHue:**(float *)*hue*
saturation:(float *)*saturation*
brightness:(float *)*brightness*
alpha:(float *)*alpha*

Returns the HSB and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. It's an error if the receiver isn't a CMYK color.

– (void)**getRed:**(float *)*red*
green:(float *)*green*
blue:(float *)*blue*
alpha:(float *)*alpha*

Returns the RGB and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. It's an error if the receiver isn't a CMYK color.

– (void)**getWhite:**(float *)*white*
alpha:(float *)*alpha*

Returns the grayscale and alpha values in the respective arguments. If NULL is passed in as an argument, the method doesn't set that value. It's an error if the receiver isn't a CMYK color.

Retrieving Individual Components

– (float)**alphaComponent**	Returns the alpha (opacity) component (1.0 by default).
– (float)**blackComponent**	Returns the black component. It’s an error if the receiver isn’t a CMYK color.
– (float)**blueComponent**	Returns the blue component. It’s an error if the receiver isn’t an RGB color.
– (float)**brightnessComponent**	Returns the brightness component of the HSB color equivalent to the receiver. It’s an error if the receiver isn’t an RGB color.
– (NSString *)**catalogNameComponent**	Returns the name of the catalog containing this color, or **nil** if the receiver’s color space isn’t NSNamedColorSpace.
– (NSString *)**colorNameComponent**	Returns the name of this color, or **nil** if the receiver’s color space isn’t NSNamedColorSpace.
– (float)**cyanComponent**	Returns the cyan component. It’s an error if the receiver isn’t a CMYK color.
– (float)**greenComponent**	Returns the green component. It’s an error if the receiver isn’t an RGB color.
– (float)**hueComponent**	Returns the hue component of the HSB color equivalent to the receiver. It’s an error if the receiver isn’t an RGB color.
– (NSString *)**localizedCatalogNameComponent**	Like **catalogNameComponent**, but returns a localized string.
– (NSString *)**localizedColorNameComponent**	Like **colorNameComponent**, but returns a localized string.
– (float)**magentaComponent**	Returns the magenta component. It’s an error if the receiver isn’t a CMYK color.
– (float)**redComponent**	Returns the red component. It’s an error if the receiver isn’t an RGB color.
– (float)**saturationComponent**	Returns the saturation component of the HSB color equivalent to the receiver. It’s an error if the receiver isn’t an RGB color.
– (float)**whiteComponent**	Returns the white component. It’s an error if the receiver isn’t a grayscale color.
– (float)**yellowComponent**	Returns the yellow component. It’s an error if the receiver isn’t a CMYK color.

Converting to Another Color Space

– (NSString *)**colorSpaceName**

Returns the name of the NSColor's color space.

– (NSColor *)**colorUsingColorSpaceName:**(NSString *)*colorSpace*

Returns a newly created NSColor whose color is the same as the receiver's, except that the new NSColor is in the color space named *colorSpace*. This method calls **colorUsingColorSpaceName:device:** with the current device, indicating that the color is appropriate for the current device (the current window if drawing, or the current printer if printing).

– (NSColor *)**colorUsingColorSpaceName:**(NSString *)*colorSpace*
device:(NSDictionary *)*deviceDescription*

Returns a newly created NSColor whose color is the same as the receiver's, except that the new NSColor is in the color space named *colorSpace* and is specific to the device described by *deviceDescription.*

Changing the Color

– (NSColor *)**blendedColorWithFraction:**(float)*fraction*
ofColor:(NSColor *)*aColor*

Returns a newly created NSColor in NSCalibratedRGBColorSpace whose component values are a weighted sum of the receiver's and *aColor*'s. The method converts *aColor* and a copy of the receiver to RGB, and then sets each component of the returned color to *fraction* of *aColor*'s value plus 1 – *fraction* of the receiver's. If the colors can't be converted to NSCalibratedRGBColorSpace, **nil** is returned.

– (NSColor *)**colorWithAlphaComponent:**(float)*alpha*

Returns a newly created NSColor that has the same color space and component values as the receiver, except that its alpha component is *alpha*. If the receiver's color space doesn't include an alpha component, the receiver is returned.

Copying and Pasting

+ (NSColor *)**colorFromPasteboard:**(NSPasteboard *)*pasteBoard*

Returns the NSColor currently on the pasteboard, or **nil** if the pasteboard doesn't contain color data. The returned color's alpha component is set to 1.0 if **ignoresAlpha** returns YES.

– (void)**writeToPasteboard:**(NSPasteboard *)*pasteBoard*

Writes the receiver's data to the pasteboard, unless the pasteboard doesn't support color data (in which case the method does nothing).

Drawing

– (void)**drawSwatchInRect:**(NSRect)*rect*

Draws the current color in the rectangle *rect*. Subclasses adorn the rectangle in some manner to indicate the type of color. This method is invoked by color wells, swatches, and other user-interface objects that need to display colors.

– (void)**set**

Sets the color of subsequent PostScript drawing to the color that the receiver represents. If the application is drawing to the screen rather than printing, this method also sets the current drawing context's alpha value to the value returned by **alphaComponent**.

NSColorList

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: AppKit/NSColorList.h

Class Description

Instances of NSColorList are used to manage named lists of NSColors. NSColorPanel's list-mode color picker uses instances of NSColorList to represent any lists of colors that come with the system, as well as any lists created by the user. An application can use NSColorList to manage document-specific color lists, which may be added to an application's NSColorPanel using its **attachColorList:** method.

An NSColorList is similar to a dictionary object: An NSColor is added to, looked up in, and removed from the list by specifying its key, which is an NSString. In addition, colors can be inserted at specified positions in the list. The list itself has a name, specified when you create the object (using either **initWithName:** or **initWithName:fromFile:**).

An NSColorList saves and retrieves its colors from files with the extension **".clr"** in directories defined by a standard search path. To access all the color lists in the standard search path, use the **availableColorLists** method; this returns an array of NSColorLists, from which you can retrieve the individual color lists by name.

NSColorList reads color list files in several different formats; it saves color lists using the archiver API.

Initializing an NSColorList

– (id)**initWithName:**(NSString *)*name*

Initializes and returns the receiver, registering it under the specified name if the name isn't in use already.

– (id)**initWithName:**(NSString *)*name*
fromFile:(NSString *)*path*

Initializes and returns the receiver, registering it under the specified name if the name isn't in use already. *path* should be the full path to the file for the color list; *name* should be the name of the file for the color list (minus the ".clr" extension).

Getting All Color Lists

+ (NSArray *)**availableColorLists**

Returns an array of all NSColorLists found in the standard color list directories. Color lists created at run time aren't included in this list unless they're saved into one of the standard color list directories.

Getting a Color List by Name

+ (NSColorList *)**colorListNamed:**(NSString *)*name*

Searches the array that's returned by **availableColorLists** and returns the NSColorList named *name*, or **nil** if no such color list exists. *name* mustn't include the ".clr" suffix.

– (NSString *)**name**

Returns the name of the NSColorList.

Managing Colors by Key

– (NSArray *)**allKeys**

Returns an array of NSString objects that contains all the keys by which the NSColors are stored in the NSColorList. The length of this array equals the number of colors, and its contents are arranged according to the ordering specified when the colors were inserted.

– (NSColor *)**colorWithKey:**(NSString *)*key*

Returns the NSColor associated with *key*, or **nil** if there is none.

– (void)**insertColor:**(NSColor *)*color*
key:(NSString *)*key*
atIndex:(unsigned)*location*

Inserts *color* at the specified location in the list (which is numbered starting with 0). If the list already contains a color with the same key at a different location, it's removed from the old location. This method posts the NSColorListChangedNotification notification to the default notification center. Raises NSColorListNotEditableException if the color list is not editable. This method posts the NSColorListChangedNotification notification to the default notification center.

– (void)**removeColorWithKey:**(NSString *)*key*

Removes the color associated with *key* from the list. This method does nothing if the list doesn't contain the key. This method posts the NSColorListChangedNotification notification to the default notification center. Raises NSColorListNotEditableException if the color list is not editable.

– (void)**setColor:**(NSColor *)*aColor*
forKey:(NSString *)*key*

Associates the specified NSColor with the key *key*. If the list already contains *key*, this method sets the corresponding color to *aColor*; otherwise, it inserts *aColor* at the end of the list.

Editing

– (BOOL)**isEditable**

Returns YES if the color list can be modified. This depends on the source of the list: If it came from a write-protected file, this method returns NO.

Writing and Removing Files

– (BOOL)**writeToFile:**(NSString *)*path*

If *path* is a directory, saves the NSColorList in a file named *listname*.clr (where *listname* is the name with which the NSColorList was initialized). If *path* includes a file name, this method saves the file under that name. If *path* is **nil**, this method saves the file as *listname*.clr in the standard location. Returns YES upon success.

– (void)**removeFile**

Deletes the file from which the list was created, unless the user doesn't own the color list. The receiver is removed from the list of available colors, but isn't released.

NSColorPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSColorPanel.h

Class Description

NSColorPanel provides a standard user interface for selecting color in an application. It provides a number of standard color selection modes, and, with the NSColorPickingDefault and NSColorPickingCustom protocols, allows an application to add its own color selection modes. It allows the user to save swatches containing frequently used colors. Once set, these swatches are displayed by NSColorPanel in any application where it is used, giving the user color consistency between applications. NSColorPanel enables users to capture a color anywhere on the screen for use in the active application, and allows dragging colors from itself into views in an application. NSColorPanel's action message is sent to the target object when the user changes the current color.

An application has only one instance of NSColorPanel, the shared instance. Invoking the **sharedColorPanel:** method returns the shared instance of NSColorPanel, instantiating it if necessary. You can also initialize an NSColorPanel for your application by invoking NSApplication's **orderFrontColorPanel** method.

You can put NSColorPanel in any application created with Interface Builder by adding the "Colors..." item from the Menu palette to the application's menu.

Color Mask and Color Modes

The color mask determines which of the color modes are enabled for NSColorPanel. This mask is set before you initialize a new instance of NSColorPanel. NSColorPanelAllModesMask represents the logical OR of the other color mask constants: it causes the NSColorPanel to display all standard color pickers. When initializing a new instance of NSColorPanel, you can logically OR any combination of color mask constants to restrict the available color modes.

Mode	Color Mask Constant
Grayscale-Alpha	NSColorPanelGrayModeMask
Red-Green-Blue	NSColorPanelRGBModeMask
Cyan-Yellow-Magenta-Black	NSColorPanelCMYKModeMask
Hue-Saturation-Brightness	NSColorPanelHSBModeMask
TIFF image	NSColorPanelCustomPaletteModeMask
Custom color lists	NSColorPanelColorListModeMask
Color wheel	NSColorPanelWheelModeMask
All of the above	NSColorPanelAllModesMask

The NSColorPanel's color mode mask is set using the class method **setPickerMask:.** The mask must be set before creating an application's instance of NSColorPanel.

When an application's instance of NSColorPanel is masked for more than one color mode, your program can set its active mode by invoking the **setMode**: method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	NSGrayModeColorPanel
Red-Green-Blue	NSRGBModeColorPanel
Cyan-Yellow-Magenta-Black	NSCMYKModeColorPanel
Hue-Saturation-Brightness	NSHSBModeColorPanel
TIFF image	NSCustomPaletteModeColorPanel
Color lists	NSColorListModeColorPanel
Color wheel	NSWheelModeColorPanel

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load a TIFF file into the NSColorPanel, then select colors from the TIFF image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide NSPopUpLists for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors. If a color panel has been used, it uses whatever mode it was in last as the default mode when NSColorPanelAllModesMask is used to initialize the NSColorPanel. Otherwise, it uses color wheel mode.

Associated Classes and Protocols

The NSColorList class provides an API for managing custom color lists. The NSColorPanel methods **attachColorList:** and **detachColorList:** let your application add and remove custom lists from the NSColorPanel's user interface.

The protocols NSColorPickingDefault and NSColorPickingCustom provide an API for adding custom color selection to the user interface. The NSColorPicker class implements the NSColorPickingDefault protocol; you can subclass NSColorPicker and implement the NSColorPickingCustom protocol in your subclass to create your own user interface for color selection.

See also: NSColorList, NSColorPickingDefault, NSColorPicker, NSColorPickingDefault protocol, NSColorPickingCustom protocol, NSColorWell

Creating the NSColorPanel

+ (NSColorPanel *)**sharedColorPanel** — Creates if necessary and returns the shared NSColorPanel.

+ (BOOL)**sharedColorPanelExists** — Returns YES if the NSColorPanel has been created already.

Setting the NSColorPanel

+ (void)**setPickerMask:**(int)*mask* — Sets the mask that determines which color selection modes are available in the color panel.

+ (void)**setPickerMode:**(int)*mode* — Sets the color picker mode.

– (NSView *)**accessoryView** — Returns the accessory view, or **nil** if there is none.

– (BOOL)**isContinuous** — Returns YES if the NSColorPanel continuously sends the action message to the target.

– (int)**mode** — Returns the mode of the NSColorPanel.

– (void)**setAccessoryView:**(NSView *)*aView* — Sets the accessory view to *aView.*

– (void)**setAction:**(SEL)*aSelector* — Sets the action message sent to the target.

– (void)**setContinuous:**(BOOL)*flag* — Sets the NSColorPanel to continuously send the action message to the target.

– (void)**setMode:**(int)*mode* — Sets the mode of the NSColorPanel.

– (void)**setShowsAlpha:**(BOOL)*flag* — Sets the NSColorPanel to show alpha values.

– (void)**setTarget:**(id)*anObject* — Sets the target of the NSColorPanel.

– (BOOL)**showsAlpha** — Returns YES if the NSColorPanel shows alpha values.

Attaching a Color List

– (void)**attachColorList:**(NSColorList *)*aColorList*
Adds the specified list of NSColors to all the color pickers in the color panel that display color lists.

– (void)**detachColorList:**(NSColorList *)*aColorList*
Removes the specified list of NSColors from all the color pickers in the color panel that display color lists.

Setting Color

+(BOOL)**dragColor:**(NSColor **)*aColor*
withEvent:(NSEvent *)*anEvent*
fromView:(NSView *)*sourceView* — Drags *aColor* into a destination view from *sourceView.*

– (float)**alpha** — Returns the NSColorPanel's current alpha value, or 1.0 (opaque) if the panel has no opacity slider.

– (NSColor *)**color** — Returns the currently displayed color.

– (void)**setColor:**(NSColor *)*aColor* — Sets the color to be displayed. This method posts the NSColorPanelChangedNotification notification with the receiving object to the default notification center.

NSColorPicker

Inherits From: NSObject

Conforms To: NSColorPickingDefault
NSObject (NSObject)

Declared In: AppKit/NSColorPicker.h

Class Description

NSColorPicker is an abstract superclass that implements the NSColorPickingDefault protocol. The NSColorPickingDefault and NSColorPickingCustom protocols define a way to add color pickers (custom user interfaces for color selection) to the NSColorPanel. The simplest way to implement a color picker is to create a subclass of NSColorPicker, instead of implementing the NSColorPickingDefault protocol in another kind of object. (To add functionality, implement the NSColorPickingCustom methods in your subclass.)

The NSColorPickingDefault protocol specification describes the details of implementing a color picker and adding it to your application's NSColorPanel; you should look there first for an overview of how NSColorPicker works. This specification is provided to document the specific behavior of NSColorPicker's methods.

Initializing an NSColorPicker

– (id)**initWithPickerMask:**(int)*aMask*
colorPanel:(NSColorPanel *)*colorPanel*

Initializes the receiver for the specified mask and color panel, caching the *colorPanel* value so it can later be returned by the **colorPanel** method.

Getting the Color Panel

– (NSColorPanel *)**colorPanel**

Returns the NSColorPanel that owns this NSColorPicker.

Adding Button Images

– (void)**insertNewButtonImage:**(NSImage *)*newImage*
in:(NSButtonCell *)*newButtonCell*

Called by the color panel to insert a new image into the specified cell. Override this method to customize *newImage* before insertion in *newButtonCell*.

– (NSImage *)**provideNewButtonImage**

Returns the button image for the color picker. The color panel will place this image in the mode button that the user uses to select this picker. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.) The default implementation looks in the color picker's bundle for a TIFF file named after the color picker's class, with the extension ".tiff".

Setting the Mode

– (void)**setMode:**(int)*mode*

Does nothing. Override to set the color picker's mode.

Using Color Lists

– (void)**attachColorList:**(NSColorList *)*colorList*

Does nothing. Override to attach a color list to a color picker.

– (void)**detachColorList:**(NSColorList *)*colorList*

Does nothing. Override to detach a color list from a color picker.

Responding to a Resized View

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

Does nothing. Override to respond to a size change.

NSColorWell

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSColorWell.h

Class Description

NSColorWell is an NSControl for selecting and displaying a single color value. An example of an NSColorWell object (or simply color well) is found in NSColorPanel, which uses a color well to display the current color selection. NSColorWell is available from the Palettes panel of Interface Builder.

An application can have one or more active NSColorWells. You can activate multiple NSColorWells by invoking the **activate:** method with NO as its argument. When a mouse-down event occurs on an NSColorWell's border, it becomes the only active color well. When a color well becomes active, it brings up the color panel also.

The **mouseDown:** method enables an instance of NSColorWell to send its color to another NSColorWell or any other subclass of NSView that implements the NSDraggingDestination protocol.

See also: NSColorPanel (class)

Drawing

– (void)**drawWellInside:**(NSRect)*insideRect* — Draws the colored area inside the color well at the location specified by *insideRect* without drawing borders.

Activating

– (void)**activate:**(BOOL)*exclusive* — Activates the NSColorWell, displays the Color panel, and makes the NSColorPanel's current color the same as its own. If *exclusive* is YES, deactivates any other NSColorWells; if NO, keeps them active.

– (void)**deactivate** — Deactivates the NSColorWell.

– (BOOL)**isActive** — Returns YES if the NSColorWell is active.

Managing Color

– (NSColor *)**color**	Returns the color of the color well.
– (void)**setColor:**(NSColor *)*color*	Sets the color of the well to *color.*
– (void)**takeColorFrom:**(id)*sender*	Changes the color of the well to that of *sender.*

Managing Borders

– (BOOL)**isBordered**	Indicates whether the color well is bordered.
– (void)**setBordered:**(BOOL)*bordered*	Places or removes a border, depending on *bordered*.

NSControl

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSControl.h

Class Description

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

Target and Action

Target objects and action methods provide the mechanism by which NSControls interact with other objects in an application. A target is an object that an NSControl has effect over. The target class defines an action method to enable its instances to respond to user input. An action method takes only one argument: the **id** of the sender. The sender may be either the NSControl that sends the action message or another object that the target should treat as the sender. When it receives an action message, a target can return messages to the sender requesting additional information about its status. NSControl's **sendAction:to:** asks the NSApplication object, NSApp, to send an action message to the NSControl's target object. The method used for this is NSApplication's **sendAction:to:from:**. You can also set the target to **nil** and allow it to be determined at run time. When the target is **nil**, the NSApplication object must look for an appropriate receiver. It conducts its search in a prescribed order, by following the responder chain until it finds an object that can respond to the message:

- It begins with the first responder in the key window and follows **nextResponder** links up the responder chain to the NSWindow object. After the NSWindow object, it tries the NSWindow's delegate.
- If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the NSWindow object and its delegate.
- Next, it tries to respond itself. If the NSApplication object can't respond, it tries its own delegate. NSApp and its delegate are the receivers of last resort.

NSControl provides methods for setting and using the target object and the action method. However, these methods require that an NSControl have an associated subclass of NSCell that provides a target and an action, such as NSActionCell and its subclasses.

Target objects and action methods demonstrate the close relationship between NSControls and NSCells. In most cases, a user interface device consists of an instance of an NSControl subclass paired with one or more instances of an NSCell subclass. Each implements specific details of the user interface mechanism. For example, NSControl's **mouseDown:** method sends a **trackMouse:inRect:ofView:untilMouseUp:** message to an NSCell, which handles subsequent mouse and keyboard events; an NSCell sends an NSControl a **sendAction:to:** message in response to particular events. NSControl's **drawRect:** method is implemented by sending a **drawWithFrame:inView:** message to the NSCell. As another example, NSControl provides methods for setting and formatting its contents; these methods send corresponding messages to NSCell, which actually owns the contents.

See the NSActionCell class specification for more on the implementation of target and action behavior.

Changing the NSCell Class

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

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

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

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

Creating New NSControls

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

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

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

If you want to be able to change the type of cell that your subclass uses (without changing the type that its superclass uses), override **setCellClass:** to store the NSCell subclass in a global variable, and modify **cellClass** to return that variable:

```
static id myStoredCellClass;

+ setCellClass:classId
{
    myStoredCellClass = classId;
}
+ (Class) cellClass
{
    return (myStoredCellClass ? myStoredCellClass : [MyCellSubclass class]);
}
```

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

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

Initializing an NSControl Object

– (id)**initWithFrame:**(NSRect)*frameRect*	Initializes a new NSControl object in *frameRect*, and attempts to create a corresponding NSCell.

Setting the Control's Cell

+ (Class)**cellClass**	Returns **nil**; overridden by subclasses.
+ (void)**setCellClass:**(Class)*factoryId*	Implemented by subclasses to set the NSCell class used.
– (id)**cell**	Returns the control's NSCell.
– (void)**setCell:**(NSCell *)*aCell*	Sets the control's NSCell to *aCell*.

Enabling and Disabling the Control

– (BOOL)**isEnabled**	Returns whether the control reacts to mouse events.
– (void)**setEnabled:**(BOOL)*flag*	Sets whether the control reacts to mouse events.

Identifying the Selected Cell

– (id)**selectedCell**	Returns the control's selected NSCell.
– (int)**selectedTag**	Returns the tag of the control's selected cell.

Setting the Control's Value

– (double)**doubleValue**	Returns the value of the control's selected cell as a **double**.
– (float)**floatValue**	Returns the value of the control's selected cell as a **float**.
– (int)**intValue**	Returns the value of the control's selected cell as a **int**.
– (void)**setDoubleValue:**(double)*aDouble*	Sets the value of the control's selected cell to *aDouble*.
– (void)**setFloatValue:**(float)*aFloat*	Sets the value of the control's selected cell to *aFloat*.
– (void)**setIntValue:**(int)*anInt*	Sets the value of the control's selected cell to *anInt*.
– (void)**setNeedsDisplay**	Set the NeedsDisplay flag.
– (void)**setStringValue:**(NSString *)*aString*	Sets the value of the control's selected cell to *aString*.
– (NSString *)**stringValue**	Returns the value of the control's selected cell as an NSString.

Interacting with Other Controls

– (void)**takeDoubleValueFrom:**(id)*sender*	Sets the receiving NSControl's selected cell to the value obtained by sending a **doubleValue** message to *sender.*
– (void)**takeFloatValueFrom:**(id)*sender*	Sets the receiving NSControl's selected cell to the value obtained by sending a **floatValue** message to *sender.*
– (void)**takeIntValueFrom:**(id)*sender*	Sets the receiving NSControl's selected cell to the value obtained by sending a **intValue** message to *sender.*
– (void)**takeStringValueFrom:**(id)*sender*	Sets the receiving NSControl's selected cell to the value obtained by sending a **stringValue** message to *sender.*

Formatting Text

– (NSTextAlignment)**alignment**	Returns the alignment of text in the control's cell.
– (NSFont *)**font**	Returns the Font used to draw text in the control's cell.
– (void)**setAlignment:**(NSTextAlignment)*mode*	Sets the alignment mode of the text in the control's cell to *mode*.

– (void)**setFont:**(NSFont *)*fontObject* — Sets the Font used to draw text in the control's cell to *fontObject*.

– (void)**setFloatingPointFormat:**(BOOL)*autoRange* **left:**(unsigned)*leftDigits* **right:**(unsigned)*rightDigits* — Sets the display format for floating point values in the control's cell

Managing the Field Editor

– (BOOL)**abortEditing** — Aborts editing of text displayed by the NSControl.

– (NSText *)**currentEditor** — Returns the object used to edit text in the control.

– (void)**validateEditing** — Validates the user's changes to editable text.

Resizing the Control

– (void)**calcSize** — Recalculates internal size information.

– (void)**sizeToFit** — Resizes the control to fit its cell.

Displaying the Control and Cell

– (void)**drawCell:**(NSCell *)*aCell* — Redraws *aCell* if it's the control's cell.

– (void)**drawCellInside:**(NSCell *)*aCell* — Redraws *aCell*'s inside if it's the control's cell.

– (void)**selectCell:**(NSCell *)*aCell* — Selects *aCell* if it's the control's cell.

– (void)**updateCell:**(NSCell *)*aCell* — Redisplays *aCell* or marks it for redisplay.

– (void)**updateCellInside:**(NSCell *)*aCell* — Redisplays the inside of *aCell* or marks it for redisplay.

Target and Action

– (SEL)**action** — Returns the NSControl's action method.

– (BOOL)**isContinuous** — Returns whether the control's NSCell continuously sends its action.

– (BOOL)**sendAction:**(SEL)*theAction* **to:**(id)*theTarget* — Has the NSApplication object send *theAction* to *theTarget*.

– (int)**sendActionOn:**(int)*mask* — Determines when the action is sent while tracking.

– (void)**setAction:**(SEL)*aSelector* — Sets the NSControl's action method to *aSelector.*

– (void)**setContinuous:**(BOOL)*flag* — Sets whether the control's NSCell continuously sends its action.

– (void)**setTarget:**(id)*anObject* Sets the NSControl's target object to *anObject*.

– (id)**target** Returns the NSControl's target object.

Assigning a Tag

– (void)**setTag:**(int)*anInt* Sets the tag of the control's NSCell to *anInt*.

– (int)**tag** Returns the tag of the control's NSCell.

Tracking the Mouse

– (void)**mouseDown:**(NSEvent *)*theEvent* Invoked when the mouse button goes down while the cursor is within the bounds of the NSControl. This method highlights the NSControl's NSCell and sends it a **trackMouse:inRect:ofView:untilMouseUp:** message. Whenever the NSCell finishes tracking the mouse (for example, because the cursor has left the cell's bounds), the cell is unhighlighted. If the mouse button is still down and the cursor reenters the bounds, the cell is again highlighted and a new **trackMouse:inRect:ofView:untilMouseUp:** message is sent. This behavior repeats until the mouse button goes up.

– (BOOL)**ignoresMultiClick** Indicates whether multiple clicks are ignored.

– (void)**setIgnoresMultiClick**:(BOOL)*flag* Sets whether multiple clicks are ignored, according to *flag*.

Methods Implemented by the Delegate

NSControl itself doesn't have a delegate. These delegate methods are declared in **NSControl.h** but are intended for subclasses, such as NSTextField and NSMatrix, that do have delegates and that allow text editing.

– (BOOL)**control:**(NSControl *)*control* **textShouldBeginEditing:**(NSText *)*fieldEditor* Sent directly by *control* to the delegate; returns YES if the NSControl should be allowed to start editing the text.

– (BOOL)**control:**(NSControl *)*control* **textShouldEndEditing:**(NSText *)*fieldEditor* Sent directly by *control* to the delegate; returns YES if the NSControl should be allowed to end its edit session.

– (void)**controlTextDidBeginEditing:**(NSNotification *)*aNotification*
Sent by the default notification center to the delegate; *aNotification* is always NSControlTextDidBeginEditingNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSControlTextDidEndEditingNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSControlTextDidChangeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

NSCStringText

Inherits From: NSText : NSView : NSResponder : NSObject

Conforms To: NSChangeSpelling, NSIgnoreMisspelledWords (NSText)
NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSCStringText.h

Class Description

The NSCStringText class declares the programmatic interface to objects that manage text using eight-bit character encodings. The encoding is the same as the default C string encoding provided by **defaultCStringEncoding** in the NSString class. NSCStringText can be used in situations where backwards compatibility with the detailed interfaces of the NEXTSTEP Text object is important. Applications that can use the interface of NSText should do so.

The NSCStringText class is unlike most other classes in the Application Kit in its complexity and range of features. One of its design goals is to provide a comprehensive set of text-handling features so that you'll rarely need to create a subclass. An NSCStringText object can (among other things):

- Control the color of its text and background.
- Control the font and layout characteristics of its text.
- Control whether text is editable.
- Wrap text on a word or character basis.
- Write text to, or read it from, a file, as either RTF or plain ASCII data.
- Display graphic images within its text.
- Communicate with other applications through the Services menu.
- Let another object, the delegate, dynamically control its properties.
- Let the user copy and paste text within and between applications.
- Let the user copy and paste font and format information between NSCStringText objects.
- Let the user check the spelling of words in its text.
- Let the user control the format of paragraphs by manipulating a ruler.

NSCStringText can deal only with eight-bit characters. Therefore, it is not able to deal with Unicode character sets, and NSCStringText can’t be fully internationalized.

Plain and Rich NSCStringText Objects

When you create an NSCStringText object directly, by default it allows only one font, line height, text color, and paragraph format for the entire text. You can set the default font used by new NSCStringText instances by sending the NSCStringText class object a **setDefaultFont:** message. Once an NSCStringText object is created, you can alter its global settings using methods such as **setFont:**, **setLineHeight:**, **setTextGray:**, and **setAlignment:**. For convenience, such an NSCStringText object will be called a *plain* NSCStringText object.

To allow multiple values for these attributes, you must send the NSCStringText object a **setRichText:YES** message. An NSCStringText object that allows multiple fonts also allows multiple paragraph formats, line heights, and so on. For convenience, such an NSCStringText object will be called a *rich* NSCStringText object.

A rich NSCStringText object can use RTF (Rich Text Format) as an interchange format. Not all RTF control words are supported: On input, an NSCStringText object ignores any control word it doesn't recognize; some of those it can read and interpret it doesn't write out. Refer to the class description of NSText for a list of the RTF control words that an NSCStringText object recognizes.

Note: An NSCStringText object writes eight-bit characters in the default C string encoding, which differs somewhat from the ANSI character set.

In an NSCStringText object, each sequence of characters having the same attributes is called a *run*. A plain NSCStringText object has only one run for the entire text. A rich NSCStringText object can have multiple runs. Methods such as **setSelFont**: and **setSelColor:** let you programmatically modify the attributes of the selected sequence of characters in a rich NSCStringText object. As discussed below, the user can set these attributes using the Font panel and the ruler.

NSCStringText objects are designed to work closely with various objects and services. Some of these—such as the delegate or an embedded graphic object—require a degree of programming on your part. Others—such as the Font panel, spelling checker, ruler, and Services menu—take no effort other than deciding whether the service should be enabled or disabled. The following sections discuss these interrelationships.

Notifying the NSCStringText Object's Delegate

Many of an NSCStringText object's actions can be controlled through an associated object, the NSCStringText object's delegate. If it implements any of the following methods, the delegate receives the corresponding message at the appropriate time:

textWillResize:
textDidResize:oldBounds:
textWillSetSel:toFont:
textWillConvert:fromFont:toFont:
textWillStartReadingRichText:
textWillFinishReadingRichText:
textWillWrite:
textDidRead:paperSize:

So, for example, if the delegate implements the **textWillConvert:fromFont:toFont:** method, it will receive notification upon the user's first attempt to change the font of the text. Moreover, depending on the method's return value, the delegate can either allow or prohibit changes to the text. See “Methods Implemented by the Delegate”. The delegate can be any object you choose, and one delegate can control multiple NSCStringText objects.

Adding Graphics to the Text

A rich NSCStringText object allows graphics to be embedded in the text. Each graphic is treated as a single (possibly large) "character": The text's line height and character placement are adjusted to accommodate the graphic "character." Graphics are embedded in the text in either of two ways: programmatically or directly through user actions. The programmatic approach is discussed first.

In the programmatic approach, you add an object—generally a subclass of NSCell—to the text. This object manages the graphic image by drawing it when appropriate. Although NSCell subclasses are commonly used, the only requirement is that the embedded object responds to these messages—see "Methods Implemented by an Embedded Graphic Object" for more information:

highlight:withFrame:inView:
drawWithFrame:inView:
trackMouse:inRect:ofView:untilMouseUp:
cellSize:
readRichText:forView:
richTextforView:

You place the graphic object in the text by sending the NSCStringText object a **replaceSelWithCell:** message.

An NSCStringText object displays a graphic in its text by sending the managing object a **drawWithFrame:inView:** message. To record the graphic to a file or to the pasteboard, the NSCStringText object sends the managing object a **richTextforView:** message. The object must then write an RTF control word along with any data (such as the path of a TIFF file containing its image data) it might need to recreate its image. To reestablish the text containing the graphic image from RTF data, an NSCStringText object must know which class to associate with particular RTF control words. You associate a control word with a class object by sending the NSCStringText class object a **registerDirective:forClass:** message. Thereafter, whenever an NSCStringText object finds the registered control word in the RTF data being read from a file or the pasteboard, it will create a new instance of the class and send the object a **readRichText:forView:** message.

An alternate means of adding an image to the text is for the user to drag an EPS or TIFF file icon directly into an NSCStringText object. The NSCStringText object automatically creates a graphic object to manage the display of the image. This feature requires a rich NSCStringText object that has been configured to receive dragged images—see the **setImportsGraphics:** method of the NSText class.

Images that have been imported in this way can be written as RTFD documents. Programmatic creation of RTFD documents is not supported in this version of OpenStep. RTFD documents use a file package, or directory, to store the components of the document (the "D" stands for "directory"). The file package has the name of the document plus a ".rtfd" extension. The file package always contains a file called TXT.rtf for the text of the document, and one or more TIFF or EPS files for the images. An NSCStringText object can transfer information in an RTFD document to a file and read it from a file—see the **writeRTFDToFile:atomically:** and **readRTFDFromFile:** methods in the NSText methods.

Cooperating with Other Objects and Services

NSCStringText objects are designed to work with the Application Kit's font conversion system. By default, an NSCStringText object keeps the Font panel updated with the font of the current selection. It also changes the font of the selection (for a rich NSCStringText object) or of the entire text (for a default NSCStringText object) to reflect the user's choices in the Font panel or menu. To disconnect an NSCStringText object from this service, send it a **setUsesFontPanel:NO** message (this method is actually implemented by NSText—the superclass).

If an NSCStringText object is a subview of an NSScrollView, it can cooperate with the NSScrollView to display and update a ruler that displays formatting information. The NSScrollView retiles its subviews to make room for the ruler, and the NSCStringText object updates the ruler with the format information of the paragraph containing the selection. The **toggleRuler:** method controls the display of this ruler. Users can modify paragraph formats by manipulating the components of the ruler.

By means of the Services menu, an NSCStringText object can make use of facilities outside the scope of its own application. By default, an NSCStringText object registers with the services system that it can send and receive RTF and plain ASCII data. If the application containing the NSCStringText object has a Services menu, a menu item is added for each service provider that can accept or return these formats. To prevent NSCStringText objects from registering for services, send the NSCStringText class object an **excludeFromServicesMenu:YES** message before any NSCStringText objects are created.

Coordinates and sizes mentioned in the method descriptions below are in PostScript units—1/72 of an inch.

Initializing a New NSCStringT ext Object

– (id)**initWithFrame:**(NSRect)*frameRect* **text:**(NSString *)*theText* **alignment:**(NSTextAlignment)*mode*	Returns a new NSCStringText object at *frameRect* initialized with the contents of *theText* and with *mode* alignment.

Modifying the Frame Rectangle

– (void)**resizeTextWithOldBounds:**(NSRect)*oldBounds* **maxRect:**(NSRect)*maxRect*	Used by the NSCStringText object to resize and redisplay itself.

Laying Out the Text

– (int)**calcLine**	Calculates line breaks.
– (BOOL)**changeTabStopAt:**(float)*oldX* **to:**(float)*newX*	Resets the position of the specified tab stop.
– (BOOL)**charWrap**	Returns whether extra long words are wrapped.
– (void *)**defaultParagraphStyle**	Returns the default paragraph style.
– (float)**descentLine**	Returns distance from base line to bottom of line.

– (void)**getMarginLeft:**(float *)*leftMargin* **right:**(float *)*rightMargin* **top:**(float *)*topMargin* **bottom:**(float *)*bottomMargin*	Gets by reference the dimensions of margins around the text.
– (void)**getMinWidth:**(float *)*width* **minHeight:**(float *)*height* **maxWidth:**(float)*widthMax* **maxHeight:**(float)*heightMax*	Given the *widthMax* and *heightMax*, calculates the minimum area needed to display the text and returns *width* and *height* by reference.
– (float)**lineHeight**	Returns height of a line of text.
– (void *)**paragraphStyleForFont:**(NSFont *)*fontId* **alignment:**(int)*alignment*	Recalculates the paragraph style based on new font *fontId* and *alignment*.
– (void)**setCharWrap:**(BOOL)*flag*	Sets whether extra long words are wrapped.
– (void)**setDescentLine:**(float)*value*	Sets the distance from the base line to the bottom of line to *value*.
– (void)**setLineHeight:**(float)*value*	Sets the height of a line of text to *value*.
– (void)**setMarginLeft:**(float)*leftMargin* **right:**(float)*rightMargin* **top:**(float)*topMargin* **bottom:**(float)*bottomMargin*	Adjusts the margins around the text.
– (void)**setNoWrap**	Disables word wrap.
– (void)**setParagraphStyle:**(void *)*paraStyle*	Sets the default paragraph style for the entire text.
– (BOOL)**setSelProp:**(NSParagraphProperty)*property* **to:**(float)*value*	Sets a paragraph *property* for one or more selected paragraphs to *value*.

Reporting Line and Position

– (int)**lineFromPosition:**(int)*position*	Converts character *position* to line number.
– (int)**positionFromLine:**(int)*line*	Converts *line* number to character position.

Reading and Writing Text

– (void)**finishReadingRichText**	Sent after the NSCStringText object reads RTF data.
– (NSTextBlock *)**firstTextBlock**	Returns a pointer to the first text block in the NSCStringText object.

– (NSRect)**paragraphRect:**(int)*paraNumber* **start:**(int *)*startPos* **end:**(int *)*endPos*	Returns the location and size of a paragraph identified by *paraNumber*; also returns the starting and ending character positions by reference.
– (void)**startReadingRichText**	Sent before the NSCStringText object begins reading RTF data.

Editing Text

– (void)**clear:**(id)*sender*	Deletes the selected text.
– (void)**hideCaret**	Removes the caret from the text display.
– (void)**showCaret**	Displays the previously hidden caret in the text display.

Managing the Selection

– (void)**getSelectionStart:**(NSSelPt *)*start* **end:**(NSSelPt *)*end*	Gets information (by reference) relating to the starting and ending character positions of the selection.
– (void)**replaceSel:**(NSString *)*aString*	Replaces the selection with *aString*.
– (void)**replaceSel:**(NSString *)*aString* **length:**(int)*length*	Replaces the selection with *length* bytes of *aString*.
– (void)**replaceSel:**(NSString *)*aString* **length:**(int)*length* **runs:**(NSRunArray *)*insertRuns*	Replaces the selection with *length* bytes of *aString*. *insertRuns* is a pointer to the current run in the run array.
– (void)**scrollSelToVisible**	Brings the selection within the frame rectangle.
– (void)**selectError**	Selects all the text.
– (void)**selectNull**	Deselects the current selection.
– (void)**setSelectionStart:**(int)*start* **end:**(int)*end*	Selects text from characters *start* through *end*.
– (void)**selectText:**(id)*sender*	Makes the receiver the first responder and selects all text.

Setting the Font

+ (NSFont *)**defaultFont**	Returns the default NSFont object for NSCStringText objects.
+ (void)**setDefaultFont:**(NSFont *)*anObject*	Makes *anObject* the default NSFont object for NSCStringText objects.
– (void)**setFont:**(NSFont *)*fontObj* **paragraphStyle:**(void *)*paragraphStyle*	Sets the NSFont object and paragraph style for all text.

– (void)**setSelFont:**(NSFont *)*fontObj*	Sets the NSFont object for the selection.
– (void)**setSelFont:**(NSFont *)*fontObj* **paragraphStyle:**(void *)*paragraphStyle*	Sets the NSFont object and paragraph style for the selection.
– (void)**setSelFontFamily:**(NSString *)*fontName*	Sets the font family for the selection.
– (void)**setSelFontSize:**(float)*size*	Sets the font size for the selection.
– (void)**setSelFontStyle:**(NSFontTraitMask)*traits*	Sets the font style for the selection.

Finding Text

– (BOOL)**findText:**(NSString *)*textPattern* **ignoreCase:**(BOOL)*ignoreCase* **backwards:**(BOOL)*backwards* **wrap:**(BOOL)*wrap*	Searches for *textPattern* in the text, starting at the insertion point. *ignoreCase* instructs the search to disregard case; *backwards* means search backwards; *wrap* means that when the search reaches the beginning or end of the text (depending on the direction), it should continue by wrapping to the end or beginning of the text.

Modifying Graphic Attributes

– (NSColor *)**runColor:**(NSRun *)*run*	Returns the color of the specified text run.
– (NSColor *)**selColor**	Returns the color of the selected text.
– (void)**setSelColor:**(NSColor *)*color*	Sets the color of the selected text.

Reusing an NSCStringText Object

– (void)**renewFont:**(NSFont *)*newFontObj* **text:**(NSString *)*newText* **frame:**(NSRect)*newFrame* **tag:**(int)*newTag*	Resets the NSCStringText object to draw different text *newText* in font *newFontId* within frame *newFrame*.
– (void)**renewFont:**(NSString *)*newFontName* **size:**(float)*newFontSize* **style:**(int)*newFontStyle* **text:**(NSString *)*newText* **frame:**(NSRect)*newFrame* **tag:**(int)*newTag*	Resets the NSCStringText object to draw different text *newText* in the font identified by *newFontName*, *newFontSize*, and *newFontStyle*. Drawing occurs within frame *newFrame*.
– (void)**renewRuns:**(NSRunArray *)*newRuns* **text:**(NSString *)*newText* **frame:**(NSRect)*newFrame* **tag:**(int)*newTag*	Resets the NSCStringText object to draw different text *newText* in *newFrame*.

Setting Window Attributes

– (BOOL)**isRetainedWhileDrawing**	Returns whether a retained window is used for drawing.
– (void)**setRetainedWhileDrawing:**(BOOL)*flag*	Allows use of a retained window when drawing.

Assigning a Tag

– (void)**setTag:**(int)*anInt*	Makes *anInt* the NSCStringText object's tag.
– (int)**tag**	Returns the NSCStringText object's tag.

Handling Event Messages

– (void)**becomeKeyWindow**	Activates the caret if selection has width of 0.
– (void)**moveCaret:**(unsigned short)*theKey*	Moves the caret in response to arrow keys.
– (void)**resignKeyWindow**	Deactivates the caret.

Displaying Graphics within the Text

+ **registerDirective:**(NSString *)*directive* **forClass:***class*	Associates an RTF control word (*directive*) with *class* (usually NSCell and subclasses); objects of this class are encoded through RTF control words in NSCStringText objects.
– (NSPoint)**locationOfCell:**(NSCell *)*cell*	Returns the location of *cell*.
– (void)**replaceSelWithCell:**(NSCell *)*cell*	Replaces the selection with cell object *cell*.
– (void)**setLocation:**(NSPoint)*origin* **ofCell:**(NSCell *)*cell*	Sets the *origin* point of *cell*.

Using the Services Menu and the Pasteboard

+ **excludeFromServicesMenu:**(BOOL)*flag*	Controls whether NSCStringText objects can register for services.
– (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*	Replaces the selection with data from pasteboard *pboard*.
– (id)**validRequestorForSendType:**(NSString *)*sendType* **returnType:**(NSString *)*returnType*	Determines which Service menu items are enabled.
– (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard* **types:**(NSArray *)*types*	Copies the selection to pasteboard *pboard*.

Setting Tables and Functions

– (const NSFSM *)**breakTable**	Returns the table defining word boundaries.
– (const unsigned char *)**charCategoryTable**	Returns the table defining character categories.
– (NSCharFilterFunc)**charFilter**	Returns the current character filter function.
– (const NSFSM *)**clickTable**	Returns the table defining double-click selection.
– (NSTextFunc)**drawFunc**	Returns the current draw function.
– (const unsigned char *)**postSelSmartTable**	Returns cut and paste table for right word boundary.
– (const unsigned char *)**preSelSmartTable**	Returns cut and paste table for left word boundary.
– (NSTextFunc)**scanFunc**	Returns the current scan function.
– (void)**setBreakTable:**(const NSFSM *)*aTable*	Sets the table defining word boundaries.
– (void)**setCharCategoryTable:**(const unsigned char *)*aTable*	Sets the table defining character categories used in the word wrap or click tables.
– (void)**setCharFilter:**(NSCharFilterFunc)*aFunction*	Makes *aFunction* the character filter function.
– (void)**setClickTable:**(const NSFSM *)*aTable*	Sets the table defining double-click selection.
– (void)**setDrawFunc:**(NSTextFunc)*aFunction*	Makes *aFunction* the function that draws the text.
– (void)**setPostSelSmartTable:**(const unsigned char *)*aTable*	Sets the cut and paste table for right word boundary.
– (void)**setPreSelSmartTable:**(const unsigned char *)*aTable*	Sets the cut and paste table for left word boundary.
– (void)**setScanFunc:**(NSTextFunc)*aFunction*	Makes *aFunction* the scan function.
– (void)**setTextFilter:**(NSTextFilterFunc)*aFunction*	Makes *aFunction* the text filter function.
– (NSTextFilterFunc)**textFilter**	Returns the current text filter function.

Printing

– (void)**adjustPageHeightNew:**(float *)*newBottom* **top:**(float)*oldTop* **bottom:**(float)*oldBottom* **limit:**(float)*bottomLimit*	Assists with automatic pagination of text.

Implemented by an Embedded Graphic Object

– (NSSize)**cellSize**	Embedded cell returns its size.
– (void)**drawWithFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*	Embedded object draws itself, including frame, within *cellFrame* in *controlView*.
– (void)**highlight:**(BOOL)*flag* **withFrame:**(NSRect)*cellFrame* **inView:**(NSView *)*controlView*	Embedded object highlights or unhighlights itself with *cellFrame* of *controlView*, depending on the value of *flag*.
– (void)**readRichText:**(NSString *)*stringObject* **forView:**(NSView *)*view*	Embedded object reads its RTF representation from *stringObject* and initializes itself.
– (NSString *)**richTextForView:**(NSView *)*view*	Embedded object stores its RTF representation within view as a string object and returns it.
– (BOOL)**trackMouse:**(NSEvent *)*theEvent* **inRect:**(NSRect)*cellFrame* **ofView:**(NSView *)*controlView* **untilMouseUp:**(BOOL)*untilMouseUp*	Embedded object implements this method to track mouse movement within tracking rectangle (*cellFrame*) and to detect mouse-up event (*untilMouseUp*).

Implemented by the Delegate

– (void)**textDidRead:**(NSCStringText *)*textObject* **paperSize:**(NSSize)*paperSize*	Lets the delegate review paper size.
– (NSRect)**textDidResize:**(NSCStringText *)*textObject* **oldBounds:**(NSRect)*oldBounds*	Reports size change to delegate.
– (NSFont *)**textWillConvert:**(NSCStringText *)*textObject* **fromFont:**(NSFont *)*font* **toFont:**(NSFont *)*font*	Lets delegate intercede in selection's font change.
– (void)**textWillFinishReadingRichText:**(NSCStringText *)*textObject*	Informs delegate that the NSCStringText object finished reading RTF data.
– (void)**textWillResize:**(NSCStringText *)*textObject*	Informs delegate of impending size change.
– (void)**textWillSetSel:**(NSCStringText *)*textObject* **toFont:**(NSFont *)*font*	Lets delegate intercede in the updating of the Font panel.
– (void)**textWillStartReadingRichText:**(NSCStringText *)*textObject*	Informs delegate that NSCStringText object will read RTF data.
– (NSSize)**textWillWrite:**(NSCStringText *)*textObject*	Lets the delegate specify paper size.

Compatibility Methods

- (NSCStringTextInternalState *)**cStringTextInternalState**

Returns a structure that represents the instance variables of the NSCStringText object. The structure is defined in **appkit/NSCStringText.h**, and in the "Types and Constants" section of the Application Kit documentation. Note that this method is provided for applications that really must depend on changing the values of an NSCStringText object's instance variables.

NSCursor

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Inherits From: AppKit/NSCursor.h

Class Description

An NSCursor holds an image that the window system can display for the cursor. An NSCursor is initialized with an NSImage object (which can subsequently be replaced by sending the NSCursor a **setImage:** message). To make the window system display a particular image as the current cursor, simply send a **set** message to the NSCursor instance associated with that image.

For automatic cursor management, an NSCursor can be assigned to a cursor rectangle within a window. When the window is key and the user moves the cursor into the rectangle, the NSCursor becomes the current cursor. It ceases to be the current cursor when the cursor leaves the rectangle. The assignment is made using NSView's **addCursorRect:cursor:** method, usually inside a **resetCursorRects** method:

```
- (void)resetCursorRects
{
    [self addCursorRect:someRect cursor:theNSCursorObject];
}
```

This is the recommended way of associating a cursor with a particular region inside a window. However, the NSCursor class provides two other ways of setting the cursor:

- The class maintains its own stack of cursors. Pushing an NSCursor instance on the stack sets it to be the current cursor. Popping an NSCursor from the stack sets the next NSCursor in line, the one that's then at the top of the stack, to be the current cursor.
- An NSCursor can be made the owner of a tracking rectangle and told to set itself when it receives a mouse-entered or mouse-exited event.

The Application Kit provides two ready-made NSCursor instances: the standard arrow cursor, and the I-beam cursor that's displayed over editable or selectable text. These can be retrieved with the class methods **arrowCursor** and **IBeamCursor**, respectively. There's no NSCursor instance for the wait cursor. The wait cursor is displayed automatically by the system, without any required program intervention.

Initializing a New NSCursor Object

– (id)**initWithImage:**(NSImage *)*newImage* Initializes a new NSCursor object with *newImage*.

Defining the Cursor

– (NSPoint)**hotSpot**	Returns the point on the cursor that's aligned with the mouse.
– (NSImage *)**image**	Returns the NSImage object that has the cursor image.
– (void)**setHotSpot:**(NSPoint)*spot*	Sets the point on the cursor that's aligned with the mouse.
– (void)**setImage:**(NSImage *)*newImage*	Makes *newImage* the NSImage object that supplies the cursor image.

Setting the Cursor

+ (void)**hide**	Hides the cursor.
+ (void)**pop**	Restores the previous cursor.
+ (void)**setHiddenUntilMouseMoves**:(BOOL)*flag*;	Hides cursor when *flag* is YES; reveals it otherwise.
+ (void)**unhide**	Shows the cursor.
– (BOOL)**isSetOnMouseEntered**	Returns YES if **mouseEntered:** sets cursor.
– (BOOL)**isSetOnMouseExited**	Returns YES if **mouseExited:** sets cursor.
– (void)**mouseEntered:**(NSEvent *)*theEvent*	Responds to a mouse-entered event by setting the cursor if **setOnMouseEntered** was sent.
– (void)**mouseExited:**(NSEvent *)*theEvent*	Responds to a mouse-exited event by setting the cursor if **setOnMouseExited** was sent.
– (void)**pop**	Removes the topmost NSCursor object from the cursor stack, and makes the next NSCursor down the current cursor.
– (void)**push**	Puts the receiving NSCursor on the cursor stack and sets it to be the current cursor.
– (void)**set**	Sets the NSCursor to be the current cursor.
– (void)**setOnMouseEntered:**(BOOL)*flag*	Determines whether **mouseEntered:** sets cursor.
– (void)**setOnMouseExited:**(BOOL)*flag*	Determines whether **mouseExited:** sets cursor.

Getting the Cursor

+ (NSCursor *)**arrowCursor**	Returns an arrow cursor.
+ (NSCursor *)**currentCursor**	Returns the current cursor.
+ (NSCursor *)**IBeamCursor**	Returns an I-beam cursor.

NSCustomImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding, NSCopying (NSImageRep)
NSObject (NSObject)

Declared In: AppKit/NSCustomImageRep.h

An NSCustomImageRep is an object that uses a delegated method to render an image. When called upon to produce the image, it sends a message to its delegate to have the method performed.

Like most other kinds of NSImageReps, an NSCustomImageRep is generally used indirectly, through an NSImage object. An NSImage must be able to choose between various representations of a given image. It also needs to provide an off-screen cache of the appropriate depth for any image it uses. It determines this information by querying its NSImageReps.

Thus to work with an NSImage, an NSCustomImageRep must be able to provide some information about its image. Use the following methods, inherited from the NSImageRep class, to set these attributes of the NSCustomImageRep:

setSize:
setColorSpaceName:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

Initializing a New NSCustomImageRep

– (id)**initWithDrawSelector:**(SEL)*aSelector* **delegate:**(id)*anObject*
Initializes a new instance so that it delegates the responsibility for drawing to *anObject*. When the NSCustomImageRep receives a **draw** message, it sends an *aSelector* message to *anObject*.

Identifying the Object

– (id)**delegate**
Returns the delegate.

– (SEL)**drawSelector**
Returns the associated draw method selector.

NSDataLink

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSDataLink.h

Class Description

An NSDataLink object (or *data link*) defines a single link between a selection in a source document and a dependent, dynamically updated selection in a destination document.

A data link is typically created when linkable data is copied to the pasteboard. First, an NSSelection object describing the data is created. Then a link to that selection is created using **initLinkedToSourceSelection:managedBy:supportingTypes:**. The link can then be written to the pasteboard using **writeToPasteboard:**. Usually, after the link has been written to the pasteboard (or saved to a file using **writeToFile:**) the link is freed because it's generally of no further use to the source application.

Once the data and link have been written to the pasteboard, they can be added to a destination document by an object that can respond to a message to Paste and Link. The object responding to this message will paste the data as usual. The destination application will then read the link from the pasteboard using **initWithPasteboard:**, create an NSSelection describing the linked data within the destination document, and will add the link by sending **addLink:at:** to the document's NSDataLinkManager object (also known as a *data link manager* or simply *link manager*).

When the link is added to the destination document's link manager, it becomes a *destination link*. At that time, the data link's object establishes a connection with the source document's link manager, which automatically creates a *source link* in the source application; the source link refers to the source selection.

A link that isn't managed by a link manager is a *broken link*. (Both source and destination links have link managers.) All links are broken links when they are created. Links can be explicitly broken (ensuring that they cause no updates) using the **break** method. Broken links (that aren't former source links) can be hooked up as destination links with the **addLink:at:** method. The disposition of a link (destination, source, or broken) can be retrieved with the **disposition** method. Most of the messages defined by the NSDataLink class can be sent to a link of any disposition, but some only make sense when sent to a link with a specific disposition; these are so noted in their method descriptions.

Links of all dispositions (except links to files) maintain an NSSelection object referring to the link's selection in the source document; this selection is returned by the **sourceSelection** method. Links directly to files represent entire files rather than selections in a document; these links are created with **initLinkedToFile:** and have no source selection.

Source and destination links also maintain an NSSelection describing the location of the data in the destination document; this selection is returned by the **destinationSelection** method.

See the NSSelection class description for more information on NSSelection objects.

Initializing a Link

– (id)**initLinkedToFile:**(NSString *)*filename* — Initializes a new instance corresponding to *filename*.

– (id)**initLinkedToSourceSelection:**(NSSelection *)*selection*
managedBy:*(NSDataLinkManager *)linkManager*
supportingTypes:(NSArray *)*newTypes* — Initializes a newly allocated instance corresponding to a selection in the source document *selection*. *linkManager* is the source document's link manager. *newTypes* is a set of types that *linkManager*'s delegate is willing to provide when a destination of the link requests the data described by *selection*.

– (id)**initWithContentsOfFile**:(NSString *)*filename* — Initializes a new instance from *filename*.

– (id)**initWithPasteboard:**(NSPasteboard *)*pasteboard* — Initializes a new instance from *pasteboard*.

Exporting a Link

– (BOOL)**saveLinkIn:**(NSString *)*directoryName* — Saves the link in a filename provided by the user; the NSSavePanel’s initial directory is in *directoryName*.

– (BOOL)**writeToFile:**(NSString *)*filename* — Writes the link into the file *filename*, returning NO if the file can’t be written.

– (void)**writeToPasteboard:**(NSPasteboard *)*pasteboard* — Writes the link onto the pasteboard *pasteboard*.

Information about the Link

– (NSDataLinkDisposition)**disposition** — Identifies the link’s type.

– (NSDataLinkNumber)**linkNumber** — Returns the link’s number.

– (NSDataLinkManager *)**manager** — Returns the link’s manager.

Information about the Link’s Source

– (NSDate *)**lastUpdateTime** — Returns the last time the link was updated.

– (BOOL)**openSource** — Opens the source document of the link and makes the source selection visible.

– (NSString *)**sourceApplicationName** — Returns the name of the application that owns the source document.

– (NSString *)**sourceFilename** Returns the file name of the source document.

– (NSSelection *)**sourceSelection** Returns the source selection.

– (NSArray *)**types** Returns the types that the source document can provide.

Information about the Link's Destination

– (NSString *)**destinationApplicationName** Returns the name of the application that owns the destination document.

– (NSString *)**destinationFilename** Returns the file name of the destination document.

– (NSSelection *)**destinationSelection** Returns the destination selection.

Changing the Link

– (BOOL)**break** Breaks the link

– (void)**noteSourceEdited** Informs a source link that the data referred to by its source selection has changed.

– (void)**setUpdateMode:**(NSDataLinkUpdateMode)*mode*
Sets the link's update mode to *mode*.

– (BOOL)**updateDestination** Updates the data referred to by the link's destination selection with the contents referred to by the source selection.

– (NSDataLinkUpdateMode)**updateMode** Returns the link's update mode.

NSDataLinkManager

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: AppKit/NSDataLinkManager.h

Class Description

An NSDataLinkManager object (also known as a *data link manager* or simply *link manager*) manages data linked from and into a document through NSDataLink objects. NSDataLink objects (or *data links*) provide a link between a selection in a source document and a dependent, dynamically updated selection in a destination document. When a user does a Paste and Link command in the destination document, the link manager creates the link in response to a **addLink:at:** message. When this link is added to the destination document, it makes a connection with the source document's link manager, which creates a source link in the source application.

If an application supports data linking, a link manager should be instantiated for every document the application creates. A link manager must be assigned a delegate that assists it in keeping the document up to date; this delegate must implement some or all of the methods listed in the "Methods Implemented by the Delegate" section of this class specification. In addition, the delegate must keep the link manager informed of the state of the document, sending it messages whenever the document is edited, saved, or otherwise altered.

Only applications that support continuously updating links need to be aware of when source links are created; these applications can have the delegate of the destination document's link manager return YES in response to a **dataLinkManagerTracksLinksIndividually:** message, and then respond to **dataLinkManager:startTrackingLink:** messages to receive notifications that source links are created.

For more information about NSDataLink objects, see the NSDataLink class description. See the NSSelection class description for more information on NSSelection objects.

Initializing and Freeing a Link Manager

– (id)**initWithDelegate:**(id)*anObject*	Initializes and returns a newly allocated instance, designating *anObject* as the delegate.
– (id)**initWithDelegate:**(id)*anObject* **fromFile:**(NSString *)*path*	Initializes and returns a newly allocated instance designating *anObject* as the delegate. The document's file is specified by the full path *path.*

Adding and Removing Links

– (BOOL)**addLink:**(NSDataLink *)*link*
  at:(NSSelection *)*selection*
Adds the link *link* to the document, indicating that the data in the document described by *selection* is dependent upon the link.

– (BOOL)**addLinkAsMarker:**(NSDataLink *)*link*
  at:(NSSelection *)*selection*
Incorporates *link* into the document as a marker in the location of the destination document described by *selection*.

– (NSDataLink *)**addLinkPreviouslyAt:**(NSSelection *)*oldSelection*
  fromPasteboard:(NSPasteboard *)*pasteboard*
  at:(NSSelection *)*selection*
Creates and adds a new destination link corresponding to the same source data as the link described by the destination selection *oldSelection* with the new link's destination selection provided in *selection;* the document's links must have been written to the pasteboard *pasteboard.*

– (void)**breakAllLinks**
Breaks all the destination links in the document.

– (void)**writeLinksToPasteboard:**(NSPasteboard *)*pasteboard*
Writes all the link manager’s links to *pasteboard*.

Informing the Link Manager of Document Status

– (void)**noteDocumentClosed**
Informs link manager that document has been closed.

– (void)**noteDocumentEdited**
Informs link manager that document has been edited.

– (void)**noteDocumentReverted**
Informs link manager that changes have been reverted.

– (void)**noteDocumentSaved**
Informs link manager that document has been saved.

– (void)**noteDocumentSavedAs:**(NSString *)*path*
Informs link manager that document has been saved in the file specified by the full pathname *path*.

– (void)**noteDocumentSavedTo:**(NSString *)*path*
Informs link manager that document has been saved in the file specified by the full pathname *path*.

Getting and Setting Information about the Link Manager

– (id)**delegate**
Returns the data link manager’s delegate.

– (BOOL)**delegateVerifiesLinks**
Returns YES if delegate is asked to verify updates.

– (NSString *)**filename**
Returns the filename for the link manager’s document.

– (BOOL)**interactsWithUser**
Tells whether the link manager displays panels if link errors occur.

– (BOOL)**isEdited** Returns YES if the document was edited since the last save.

– (void)**setDelegateVerifiesLinks:**(BOOL)*flag* Sets whether the delegate is asked to verify updates.

– (void)**setInteractsWithUser:**(BOOL)*flag* Sets whether the link manager displays panels if link errors occur.

Getting and Setting Information about the Manager’s Links

– (BOOL)**areLinkOutlinesVisible** Returns YES if outlines are visible.

– (NSEnumerator *)**destinationLinkEnumerator** Returns an enumerator of the destination’s source links.

– (NSDataLink *)**destinationLinkWithSelection:**(NSSelection *)*destSel*
Returns the destination link for the selection *destSel*.

– (void)**setLinkOutlinesVisible:**(BOOL)*flag* Sets whether outlines are visible.

– (NSEnumerator *)**sourceLinkEnumerator** Returns an enumerator of the receiver’s source links.

Methods Implemented by the Delegate

– (BOOL)**copyToPasteboard:**(NSPasteboard *)*pasteboard*
at:(NSSelection *)*selection*
cheapCopyAllowed:(BOOL)*flag*
Implemented by the link manager’s delegate to supply the source data described by *selection* on the pasteboard *pasteboard*. If *flag* is YES, the system guarantees that no events will be processed by the application before the delegate is requested to provide the specified data; in this case, the application doesn’t necessarily have to write any data representations to the pasteboard. This method should return YES upon success, or NO if the selection can’t be resolved.

– (void)**dataLinkManager:**(NSDataLinkManager *)*sender*
didBreakLink:(NSDataLink *)*link*
Informs the delegate that the destination link *link* was broken and thus data based on the link's destination selection will no longer be updated.

– (BOOL)**dataLinkManager:**(NSDataLinkManager *)*sender*
isUpdateNeededForLink:(NSDataLink *)*link*
Returns YES if the source data identified by *link*'s source selection has been modified since the link's last update time.

– (void)**dataLinkManager:**(NSDataLinkManager *)*sender*
startTrackingLink:(NSDataLink *)*link*
Informs the delegate that a destination document has established a data link *link* to the link manager's document and is tracking it.

– (void)**dataLinkManager:**(NSDataLinkManager *)*sender*
stopTrackingLink:(NSDataLink *)*link*

Informs the delegate that a destination is no longer tracking the source link *link*.

– (void)**dataLinkManagerCloseDocument:**(NSDataLinkManager *)*sender*

Closes documents opened without the user interface.

– (void)**dataLinkManagerDidEditLinks:**(NSDataLinkManager *)*sender*

Informs the delegate that link data has been modified; the delegate should use this notification to mark the document as edited.

– (void)**dataLinkManagerRedrawLinkOutlines:**(NSDataLinkManager *)*sender*

Directs the delegate to redraw objects with link outlines.

– (BOOL)**dataLinkManagerTracksLinksIndividually:**(NSDataLinkManager *)*sender*

Returns whether the receiver is willing to track links individually.

– (BOOL)**importFile:**(NSString *)*filename*
at:(NSSelection *)*selection*

Imports the file *filename* at the destination described by *selection*. Returns YES upon success, or NO if the selection can't be resolved.

– (BOOL)**pasteFromPasteboard:**(NSPasteboard *)*pasteboard*
at:(NSSelection *)*selection*

Pastes the updated data that has been made available on *pasteboard*. The destination for the data is described by *selection*, which was supplied to the link manager as an argument to the **addLink:at:** method. Returns YES upon success, or NO if the selection can't be resolved.

– (BOOL)**showSelection:**(NSSelection *)*selection*

Shows the source data for the specified selection *selection*. Returns YES upon success, or NO if the selection can't be resolved.

– (NSWindow *)**windowForSelection:**(NSSelection *)*selection*

Returns the NSWindow object for the given *selection*, or **nil** if the selection can't be resolved.

NSDataLinkPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSDataLinkPanel.h

Class Description

An NSDataLinkPanel is an NSPanel that allows the user to inspect data links. The NSDataLinkPanel functions primarily by sending messages to the current data link manager (representing the current document) and to the current link (representing the current selection if it's based on a data link). Thus, the panel should be informed, by a **setLink:manager:isMultiple:** message, any time the selection changes or a document is created or activated. Since the selection may need to be tracked even before the panel is created, this message can be sent to either the NSDataLinkPanel class or the shared instance.

The NSDataLinkPanel is generally displayed using NSApplication's **orderFrontDataLinkPanel:** method. An application's sole instance of NSDataLinkPanel can be accessed with the **sharedDataLinkPanel** method.

Initializing

+ (NSDataLinkPanel *)**sharedDataLinkPanel**	Initializes and returns the shared NSDataLinkPanel object.

Keeping the Panel Up to Date

+ (void)**getLink:**(NSDataLink **)*link* **manager:**(NSDataLinkManager **)*linkManager* **isMultiple:**(BOOL *)*flag*	Gets information about the NSDataLinkPanel's currently selected link; returns the link in *link*, the link manager in *linkManager*, and the multiple selection status in *flag*.
+ (void)**setLink:**(NSDataLink *)*link* **manager:**(NSDataLinkManager *)*linkManager* **isMultiple:**(BOOL)*flag*	Informs the receiver of the current document and selection using *link* as the currently selected link and *linkManager* as the current link manager. *flag* is YES if the panel will indicate that more than one link is selected. Returns **self**.
– (void)**getLink:**(NSDataLink **)*link* **manager:**(NSDataLinkManager **)*linkManager* **isMultiple:**(BOOL *)*flag*	Gets information about the NSDataLinkPanel's currently selected link; returns the link in *link*, the link manager in *linkManager*, and the multiple selection status in *flag*.

– (void)**setLink:**(NSDataLink *)*link*
manager:(NSDataLinkManager *)*linkManager*
isMultiple:(BOOL)*flag*

Informs the receiver of the current document and selection using *link* as the currently selected link and *linkManager* as the current link manager. *flag* is YES if the panel will indicate that more than one link is selected. Returns **self**.

Customizing the Panel

– (NSView *)**accessoryView**

Returns the NSDataLinkPanel’s custom accessory view.

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

Adds *aView* to the NSDataLinkPanel’s view hierarchy.

Responding to User Input

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

Invoked when the user clicks the Break All Links button; puts up an attention panel to confirm the user’s action, and then sends a **breakAllLinks** message to the current link manager.

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

Invoked when the user clicks the Break Link button; puts up an attention panel to confirm the user’s action, and then sends a **break** message to the current link.

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

Invoked when the user clicks the Open Source button; sends an **openSource** message to the current link.

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

Invoked when the user clicks Update from Source button; sends a message to the current link to verify and update the data source and then update the destination data. Returns **self**.

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

Invoked when the user selects the update mode; sends a **setUpdateMode:** message to the current link.

NSEPSImageRep

Inherits From: NSImageRep : NSObject

Conforms To: NSCoding, NSCopying (NSImageRep)
NSObject (NSObject)

Declared In: AppKit/NSEPSImageRep.h

Class Description

An NSEPSImageRep is an object that can render an image from encapsulated PostScript code (EPS).

Like most other kinds of NSImageReps, an NSEPSImageRep is generally used indirectly, through an NSImage object. An NSImage must be able to choose between various representations of a given image. It also needs to provide an off-screen cache of the appropriate depth for any image it uses. It determines this information by querying its NSImageReps.

Thus to work with an NSImage, an NSEPSImageRep must be able to provide some information about its image. The size of the object is set from the bounding box specified in the EPS header comments. Use these methods, inherited from the NSImageRep class, to set the other attributes of the NSEPSImageRep:

setColorSpaceName:
setAlpha:
setPixelsHigh:
setPixelsWide:
setBitsPerSample:

Initializing a New Instance

+ (id)**imageRepWithData**:(NSData *)*epsData*	Invokes **initWithData:** to return an instance with data from *epsData*.
– (id)**initWithData:**(NSData *)*epsData*	Initialize an instance with data from *epsData*.

Getting Image Data

– (NSRect)**boundingBox**	Returns the rectangle that bounds the image.
– (NSData *)**EPSRepresentation**	Returns the EPS representation of the image.

Drawing the Image

– (void)**prepareGState**

Implemented by subclasses to initialize the graphics state before the image is drawn.

NSEvent

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSEvent.h

Class Description

An NSEvent object contains information about an event such as a mouse-click or a key-down. The window system associates each such user action with a window, reporting the event to the application that created the window. Pertinent information about each event—such as which character was typed and where the mouse was located—is collected in an NSEvent object and made available to the application. As events are received in the application, they're temporarily placed in storage called the event queue. When the application is ready to process an event, it takes an NSEvent from the queue.

NSEvents are typically passed to the responder chain—a set of objects within the window that inherit from NSResponder. For example, NSResponder's **mouseDown:** and **keyDown:** methods take an NSEvent as an argument. When an NSApplication retrieves an NSEvent from the event queue, it dispatches it to the appropriate NSWindow (which is itself an NSResponder) by invoking **keyDown:** or a similar message. The NSWindow in turn passes the event to the first responder, and the event gets passed on down the responder chain until some object handles it. In the case of a mouse-down, a **mouseDown:** message is sent to the NSView in which the user clicked the mouse, which relays the message to its next responder if it can't handle the message itself.

Most events follow this same path: from the window system to the application's event queue, and from there, to the appropriate objects of the application. However, the Application Kit can create an NSEvent from scratch and insert it into the event queue for distribution, or send it directly to its destination. (It's rare for an *application* to create an event directly, but it's possible, using NSEvent class methods. The newly created events can be added to the event queue by invoking NSWindow's (or NSApplication's) **postEvent:atStart:** method.

Events are retrieved from the event queue by calling the NSWindow method **nextEventMatchingMask:untilDate:inMode:dequeue:** or a similar NSApplication method. These methods return an instance of NSEvent. The nature of the retrieved event can then be ascertained by invoking NSEvent instance methods—**type**, **window**, and so forth. All types of events are associated with a window. The corresponding NSWindow object can be gotten by invoking **window**. The location of the event within the window's coordinate system is obtained from **locationInWindow**, and the time of the event is gotten from **timestamp**. The **modifierFlags** method returns an indication of which modifier keys (Command, Control, Shift, and so forth) were held down while the event occurred.

The **type** method returns an NSEventType, a constant that identifies the sort of event. The different types of events fall into five groups:

- Keyboard events
- Mouse events
- Tracking-rectangle events
- Periodic events
- Cursor-update events

Some of these groups comprise several NSEventType constants; others only one. The following sections discuss the groups, along with the corresponding NSEventType constants.

Keyboard Events

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

- NSKeyDown: The user generated a character by pressing a key.
- NSKeyUp: The key was released.
- NSFlagsChanged: The user pressed or released a modifier key, or turned Alpha Lock on or off.

Of these, key-down events are the most useful to the application. When the **type** method returns NSKeyDown, your next step is typically to determine the character or characters generated by the key-down, by sending the NSEvent a **characters** message.

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

Mouse Events

Mouse events are generated by changes in the state of the mouse buttons and by changes in the position of the mouse cursor on the screen. This category consists of:

- NSLeftMouseDown, NSLeftMouseUp, NSRightMouseDown, NSRightMouseUp: Two sets of mouse-down and mouse-up events, one for the left mouse button and one for the right. "Mouse-down" means the user pressed the button; "mouse-up" means the button was released. If the mouse has just one button, only left mouse events are generated. By sending a **clickCount** message to the NSEvent, you can determine whether the mouse event was a single-click, double-click, and so on.
- NSLeftMouseDragged, NSRightMouseDragged: Two types of mouse-dragged events—one for when the mouse is moved with its left mouse button down, or with both buttons down, and one for when it's moved with just the right button down. A mouse with a single button generates only left mouse-dragged events. As the mouse is moved with a button down, a series of mouse-dragged events is produced. The series is always preceded by a mouse-down event and followed by a mouse-up event.
- NSMouseMoved: The user moved the mouse without holding down either mouse button.

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

Note: OpenStep doesn’t specify facilities for the third button of a three-button mouse.

Tracking-Rectangle Events

NSMouseEntered and NSMouseExited events are like the “Mouse Events” listed previously, in that they’re dependent on mouse movements. However, unlike the others, they’re generated only if the application has asked the window system to set a tracking rectangle in a window. An NSMouseEntered or NSMouseExited event is created when the cursor has entered the tracking rectangle or left it. A window can have any number of tracking rectangles; the NSEvent method **trackingNumber** identifies which rectangle was entered or exited.

Periodic Events

An event of type NSPeriodic simply notifies an application that a certain time interval has elapsed. By using the NSEvent class method **startPeriodicEventsAfterDelay:withPeriod:**, an application can register that it wants periodic events and that they should be placed in its event queue at a certain frequency. When the application no longer needs them, the flow of periodic events can be turned off by invoking **stopPeriodicEvents**. An application can’t have more than one stream of periodic events active at a time. Unlike keyboard and mouse events, periodic events aren’t dispatched to an NSWindow.

Cursor-Update Events

Events of type NSCursorUpdate are used to implement NSView’s cursor-rectangle methods. An NSCursorUpdate event is generated when the cursor has crossed the boundary of a predefined rectangular area. The application can respond by updating the cursor's shape.

Creating NSEvent Objects

+ (NSEvent *)**enterExitEventWithType:**(NSEventType)*type*
location:(NSPoint)*location*
modifierFlags:(unsigned int)*flags*
timestamp:(NSTimeInterval)*time*
windowNumber:(int)*windowNum*
context:(NSDPSContext *)*context*
eventNumber:(int)*eventNum*
trackingNumber:(int)*trackingNum*
userData:(void *)*userData*

Returns an NSEvent object initialized with general event data and information specific to mouse tracking (*eventNum, trackingNum, userData).*

+ (NSEvent *)**keyEventWithType:**(NSEventType)*type*
location:(NSPoint)*location*
modifierFlags:(unsigned int)*flags*
timestamp:(NSTimeInterval)*time*
windowNumber:(int)*windowNum*
context:(NSDPSContext *)*context*
characters:(NSString *)*keys*
charactersIgnoringModifiers:(NSString *)*ukeys*
isARepeat:(BOOL)*repeatKey*
keyCode:(unsigned short)*code*

Returns an NSEvent object initialized with general event data and information specific to keyboard events (*keys, repeatKey, code*, *ukeys*). (*ukeys* sets the unmodified character string.)

+ (NSEvent *)**mouseEventWithType:**(NSEventType)*type*
location:(NSPoint)*location*
modifierFlags:(unsigned int)*flags*
timestamp:(NSTimeInterval)*time*
windowNumber:(int)*windowNum*
context:(NSDPSContext *)*context*
eventNumber:(int)*eventNum*
clickCount:(int)*clickNum*
pressure:(float)*pressureValue*

Returns an NSEvent object initialized with general event data and information specific to mouse events (*eventNum, clickNum, pressureValue)*.

+ (NSEvent *)**otherEventWithType:**(NSEventType)*type*
location:(NSPoint)*location*
modifierFlags:(unsigned int)*flags*
timestamp:(NSTimeInterval)*time*
windowNumber:(int)*windowNum*
context:(NSDPSContext *)*context*
subtype:(short)*subType*
data1:(int)*data1*
data2:(int)*data2*

Returns an NSEvent object initialized with general event data and information specific to kit-defined events (*subType, data1, data2)*.

Getting General Event Information

– (NSDPSContext *)**context**
Returns the Display PostScript context of the event.

– (NSPoint)**locationInWindow**
Returns the event's location in the base coordinate system of its window.

– (unsigned int)**modifierFlags**
Returns an integer bitfield containing modifier-key flags.

– (NSTimeInterval)**timestamp**
Returns the time the event occurred in seconds since system startup.

– (NSEventType)**type**
Returns the type of the event (left-mouse-up, right-mouse-dragged, key-down, etc.).

– (NSWindow *)**window**
Returns the window object associated with the event.

– (int)**windowNumber**
Returns the number of the window associated with the event.

Getting Key Event Information

– (NSString *)**characters**	Returns the character code (a string of characters generated by the key event).
– (NSString *)**charactersIgnoringModifiers**	Returns the string of characters generated by the key event as if no modifier key had been pressed (except for Shift).
– (BOOL)**isARepeat**	Returns whether the key event is being repeated (user is holding down the key).
– (unsigned short)**keyCode**	Returns the code that maps to a key on the keyboard.

Getting Mouse Event Information

– (int)**clickCount**	Returns the number of mouse clicks associated with the mouse event.
– (int)**eventNumber**	Returns the event number of the latest mouse-down event. This information is also useful for handling tracking events.
– (float)**pressure**	Returns a value indicating the pressure applied to the input device (used for appropriate devices, not mice).

Getting Tracking Event Information

– (int)**trackingNumber**	Returns the number that identifies the tracking rectangle.
– (void *)**userData**	Returns data arbitrarily associated with the event.

Requesting Periodic Events

+ (void)**startPeriodicEventsAfterDelay:**(NSTimeInterval)*delaySeconds* **withPeriod:**(NSTimeInterval)*periodSeconds*	Start generating periodic events with frequency *periodSeconds* after delay *delaySeconds* for current thread.
+ (void)**stopPeriodicEvents**	Stop generating periodic events for current thread, and discard any periodic events remaining in the queue.

Getting Information about Specially Defined Events

– (int)**data1**	Returns special data associated with the event.
– (int)**data2**	Returns special data associated with the event.
– (short)**subtype**	Returns the identifier of the specially defined event.

NSFont

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSFont.h

Class Description

The NSFont class declares the programmatic interface to objects that correspond to fonts. NSFont is in principle an abstract class that represents fonts in general, not just PostScript fonts. In practice, at this time, NSFont objects represent PostScript fonts. Each NSFont object records a font's name, size, style, and matrix. When an NSFont object receives a **set** message, it establishes its font as the current font in the PostScript Server's current graphics state.

For a given application, only one NSFont object is created for a particular PostScript font/size or font/matrix combination. That is—if you ask for 24-point Optima, a new font object is created for 24-point Optima if such an object doesn't exist already. When the NSFont class object receives a message to create a new object for a particular font, it first checks whether an object has already been created for that font. If so, the the NSFont class object returns the existing font object; otherwise, the the NSFont class object creates a new font object and returns it.

This sharing of NSFont objects minimizes the number of distinct font objects created. It also implies that no one object in your application can know whether it has the only reference to a particular NSFont object. Thus, NSFont objects shouldn't be deallocated, but should be treated like auto-released Foundation class objects.

Where *matrix* is used, it refers to a PostScript-style six-element array of numbers that indicate transformations to be applied to a font. An NSFontIdentityMatrix identifies a font matrix used for fonts created by specifying a size.

The *size* of a font in the method definitions is defined in "points", which in currently accepted practice, are actually PostScript units—a PostScript unit being defined as 1/72 of an inch, or 0.0139 of an inch. In metric equivalents, a PostScript unit is 0.3528 millimetres. PostScript "points" are minimally different from "printer's points", so for all intents and purposes you can think of PostScript units and points as interchangeable.

In general, you instantiate an NSFont object by sending one of the methods listed in "Creating a Font Object" to the NSFont class object. The methods with **system** and **user** in their names obtain special pre-determined fonts defined at the system level and the application level, respectively. In general, you would use the **fontWithName:size:** and **fontWithName:matrix:** methods to obtain a named font.

A variety of methods are available for querying a font object. In particular, AFM (Adobe Font Metrics) data can be obtained by invoking **afmDictionary** or **afmFileContents**.

Methods whose descriptions state "Returns…and matrix NSFontIdentityMatrix" actually return an NSFontIdentityMatrix whose first and fourth elements are multiplied by the current size of the font.

Exceptions

Methods listed in "Creating a Font Object" can all raise a NSFontUnavailableException if the requested font can't be constructed.

Creating a Font Object

+ (NSFont *)**boldSystemFontOfSize:**(float)*fontSize* Returns the font object representing the bold system font of size *fontSize* and matrix NSFontIdentityMatrix.

+ (NSFont *)**fontWithName:**(NSString *)*fontName* **matrix:**(const float *)*fontMatrix*
Returns a font object for font *fontName* and matrix *fontMatrix*.

+ (NSFont *)**fontWithName:**(NSString *)*fontName* **size:**(float)*fontSize*
Returns a font object for font *fontName* of size *fontSize*.

+ (NSFont *)**systemFontOfSize:**(float)*fontSize* Returns the font object representing the system font of size *fontSize* and matrix NSFontIdentityMatrix.

+ (NSFont *)**userFixedPitchFontOfSize:**(float)*fontSize*
Returns the font object representing the application's fixed-pitch font of size *fontSize* and matrix NSFontIdentityMatrix.

+ (NSFont *)**userFontOfSize:**(float)*fontSize* Returns the font object representing the application's standard font of size *fontSize* and matrix NSFontIdentityMatrix.

Setting the Font

+ (void)**setUserFixedPitchFont:**(NSFont *)*aFont* Sets the fixed-pitch font used by default in the application to *aFont*.

+ (void)**setUserFont:**(NSFont *)*aFont* Sets the standard font used by default in the application to *aFont*.

+ (void)**useFont:**(NSString *)*fontName* Registers that *fontName* is used in the document. This information is used by the printing machinery

– (void)**set** Makes this font the graphic state's current font.

Querying the Font

– (NSDictionary *)**afmDictionary** Returns the font's AFM dictionary if the font has an AFM file. The return value can possibly be **nil**, so you must check to determine if a non-**nil** value was actually returned.

– (NSString *)**afmFileContents** Returns the raw contents of the entire AFM file, in terms of strings, if the font has an AFM file. If the font does not have an AFM file, this method returns **nil**.

– (NSRect)**boundingRectForFont** Returns the bounding rectangle for the font. This is the font's FontBBox field scaled to the current size of the font.

– (NSString *)**displayName** Returns the full name of the font as displayed in the font panel. This is the localized version of the font's name. It is not necessarily the FullName field of the font.

– (NSString *)**familyName** Returns the name of the font's family.

– (NSString *)**fontName** Returns the name of the font.

– (BOOL)**isBaseFont** Indicates whether the font is a base font, as opposed to a composite font.

– (const float *)**matrix** Returns a pointer to an array of six floats representing the font's matrix. You should not alter the data pointed to by matrix. If you wish to change values for any reason you must make a copy of the matrix

– (float)**pointSize** Returns the size of the font in points.

– (NSFont *)**printerFont** Returns the printer font for the font, if the receiving font object is a screen font. Else, this method returns **self**.

– (NSFont *)**screenFont** Returns the screen font for the font, if there is one. Else this method returns **self**.

– (float)**widthOfString:**(NSString *)*string* Returns the width of *string* in the font. Use this method with caution: it assumes that the characters in *string*can all actually be rendered in the font. It uses lossy encoding methods in NSString to get the character data.

– (float *)**widths** Returns a pointer to an array representing the widths of the glyphs in the font.

Manipulating Glyphs

– (NSSize)**advancementForGlyph:**(NSGlyph)*aGlyph*

Returns the horizontal and vertical advancement for *aGlyph*. That is, this method returns the amount by which the current point would be displaced in both *x* and *y* if the specified glyph were rendered in the current font and size. In general, the *y* component of the displacement for "Western" fonts will be zero.

– (NSRect)**boundingRectForGlyph:**(NSGlyph)*aGlyph*

Returns a bounding rectangle for *aGlyph*, scaled to the font's actual size and matrix.

– (BOOL)**glyphIsEncoded:**(NSGlyph)*aGlyph*

Indicates whether *aGlyph* is encoded. That is, *aGlyph* is present in the encoding for the font.

– (NSPoint)**positionOfGlyph:**(NSGlyph)*curGlyph* **precededByGlyph:**(NSGlyph)*prevGlyph* **isNominal:**(BOOL *)*nominal*

Returns *curGlyph*'s position when it follows *prevGlyph. nominal* is a pointer to a BOOL. If not **nil**, this method fills in *nominal* with YES, to indicate that the position has been modified by kerning information, and NO to indicate that no kerning information was present.

NSFontManager

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSFontManager.h

Class Description

NSFontManager declares the programmatic interface to objects that manage font conversion in an application. NSFontManager is the center of activity for font conversion. NSFontManager accepts messages from font conversion user-interface objects such as the Font menu or the Font panel (see NSFontPanel for more details) and appropriately converts the current font in the selection by sending a **changeFont:** message up the responder chain.

When an object receives a **changeFont:** message, it should message NSFontManager (by sending it a **convertFont:** message), asking it to convert the font in whatever way the user has specified. Thus, any object containing a font that can be changed should respond to the **changeFont:** message by sending a **convertFont:** message back to the NSFontManager for each font in the selection.

To use NSFontManager, you simply insert a Font menu into your application's menu using the appropriate interface construction tools (such as Interface Builder). You can also obtain a Font menu by sending a **getFontMenu:** message to NSFontManager and then inserting the menu it returns into the application's main menu. Once the Font menu is installed, your application automatically gains the functionality of both the Font menu and the Font panel.

NSFontManager's delegate can restrict which font names will appear in the Font Panel. See "Methods Implemented by the Delegate" at the end of this class specification for more information.

NSFontManager can be used to convert a font or find out the attributes of a font. It can also be overridden to convert fonts in some application-specific manner. The default implementation of font conversion is very conservative: The font isn't converted unless all traits of the font can be maintained across the conversion.

Generally, you obtain an instance of NSFontManager by sending a **sharedFontManager** message to the NSFontManager class object. NSFontManager will return a font manager object that is shared within your application. NSFontManager normally returns a pre-defined font manager object, but the actual object which is returned can be changed by previously invoking the **setFontManagerFactory** factory to some other kind of object.

Font Traits

Fonts work mainly in terms of *traits*, or characteristics, such as bold, italic, condensed, and so on. Traits are described by a collection of constants such as **NSItalicFontMask**, **NSBoldFontMask**, and so on. The full complement of traits are defined in AppKit/NSFontManager.h. The values of traits are defined in bitwise form so they can be or'ed together, although some traits, such as **NSBoldFontMask** and **NSUnboldFontMask** naturally conflict and have the effect of turning each other off. You use one of the **convertFont**… methods to obtain a font of the desired characteristics from an existing font.

The **convertFont:toHaveTrait:** and the **convertFont:toNotHaveTrait:** methods deal with only one trait at a time. To convert a font to have (or not have) multiple traits, you must invoke these methods for each separate trait you wish to add to or remove from the font. Alternatively, use the **fontWithFamily:traits:weight:size:** method to specify multiple traits in one invocation.

The *size* of a font in the method definitions below is defined in "points", which, in the current milieu, are actually PostScript units—a PostScript unit being defined as 1/72 of an inch, or 0.0139 of an inch. In metric equivalents, a PostScript unit is 0.3528 millimetres. PostScript "points" are minimally different from "printer's points", so for all intents and purposes you can think of PostScript units and points as interchangeable.

The *weight* of a font as used in these methods is simply a value representing a point in a continuum of font weights from lightest to heaviest. There's no simple one-to-one mapping of some integer value to, say, a **bold** weight. If you query the font for its weight value, increment the value, and use it as a new weight, you'll not necessarily obtain a different face (such as a transition from medium to bold) in a new instance of the font.

Managing the FontManager

+ (void)**setFontManagerFactory:**(Class)*classId* — Sets the class used to create the NSFontManager.

+ (void)**setFontPanelFactory:**(Class)*classId* — Sets the class used to create the FontPanel.

+ (NSFontManager *)**sharedFontManager** — Returns a shared FontManager.

Converting Fonts

– (NSFont *)**convertFont:**(NSFont *)*fontObject* — Converts *fontObject* according to the user's selections from the Font panel or the Font menu.

– (NSFont *)**convertFont:**(NSFont *)*fontObject* **toFamily:**(NSString *)*family* — Returns a Font object whose traits are the same as those of *fontObject* except as specified by *family*.

– (NSFont *)**convertFont:**(NSFont *)*fontObject* **toFace:**(NSString *)*typeface* — Returns a Font object whose traits are the same as those of *fontObject* except as specified by *typeface*.

– (NSFont *)**convertFont:**(NSFont *)*fontObject* **toHaveTrait:**(NSFontTraitMask)*trait* — Returns a Font object whose traits are the same as those of *fontObject* except as altered by the addition of the traits specified by *trait*.

– (NSFont *)**convertFont:**(NSFont *)*fontObject* **toNotHaveTrait:**(NSFontTraitMask)*trait* — Returns a Font object whose traits are the same as those of *fontObject* except as altered by the removal of the traits specified by *trait*.

– (NSFont *)**convertFont:**(NSFont *)*fontObject* **toSize:**(float)*size* — Returns a Font object whose traits are the same as those of *fontObject* except as specified by *size*.

– (NSFont *)**convertWeight:**(BOOL)*upFlag* **ofFont:**(NSFont *)*fontObject* — Attempts to increase (if *upFlag* is YES) or decrease (if *upFlag* is NO) the weight of the font specified by *fontObject*.

– (NSFont *)**fontWithFamily:**(NSString *)*family* **traits:**(NSFontTraitMask)*traits* **weight:**(int)*weight* **size:**(float)*size*
Tries to find a font that matches the specified characteristics.

Setting and Getting Parameters

– (SEL)**action**
Gets the action sent by the FontManager.

– (NSArray *)**availableFonts**
Provides an array listing all available fonts.

– (NSMenu *)**fontMenu:**(BOOL)*create*
Returns the Font menu, creating one if it doesn't exist and *create* is YES.

– (NSFontPanel *)**fontPanel:**(BOOL)*create*
Returns the Font panel, creating one if it doesn't exist and *create* is YES.

– (BOOL)**isEnabled**
Returns whether the Font panel and menu are enabled.

– (BOOL)**isMultiple**
Returns whether the selection contains multiple fonts.

– (NSFont *)**selectedFont**
Returns the first font in the current selection

– (void)**setAction:**(SEL)*aSelector*
Sets the action to that specified by *aSelector* to be sent by the FontManager when the user selects a new font.

– (void)**setEnabled:**(BOOL)*flag*
Enables or disables the Font panel and menu depending on *flag*.

– (void)**setFontMenu:**(NSMenu *)*newMenu*
Sets the font menu to *newMenu*.

– (void)**setSelectedFont:**(NSFont *)*fontObject* **isMultiple:**(BOOL)*flag*
Notifies FontManager of the selection's current font from *fontObject* with *flag* indicating whether the selection has multiple fonts.

– (NSFontTraitMask)**traitsOfFont:**(NSFont *)*fontObject*
Returns the font traits of *fontObject*.

– (int)**weightOfFont:**(NSFont *)*fontObject*
Returns the font weight of *fontObject*.

Target and Action Methods

– (BOOL)**sendAction**
Dispatches the action message up the responder chain.

Assigning a Delegate

– (id)**delegate**
Returns the FontManager's delegate.

– (void)**setDelegate:**(id)*anObject*
Sets the FontManager's delegate to *anObject*.

Methods Implemented by the Delegate

– (BOOL)**fontManager:**(id)*sender* **willIncludeFont:**(NSString *)*fontName*

Responds to a message informing the FontManager's delegate that the FontPanel is about to include *fontName* in the list displayed to the user; if this method returns NO, the font isn't added; otherwise, it is.

NSFontPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSFontPanel.h

Class Description

The NSFontPanel class declares the programmatic interface to a user-interface object that displays a list of available fonts, enabling users to preview them and change the typefaces in which text is displayed. Actual changes to text are effected through conversion messages sent to the NSFontManager. There is only one NSFontPanel object for each application.

In general, you add the facilities of the NSFontPanel (and of the other components of the font conversion system: the NSFontManager and the Font menu) to your application through interface construction tools (such as Interface Builder). You do this by including a Font menu into one of your application's menus. At runtime, when the user chooses the Font Panel command for the first time, the NSFontPanel object is created and hooked into the font conversion system. You can also create (or access) NSFontPanel through the **sharedFontPanel** method.

An NSFontPanel can be customized by adding an additional NSView object or hierarchy of NSView objects by using the **setAccessoryView:** method. If you want the NSFontManager to instantiate a panel object from some class other than NSFontPanel, use the NSFontManager's **setFontPanelFactory:** method. See NSFontManager for details on the font manager object that performs font conversion tasks.

Creating an NSFontPanel

+ (NSFontPanel *)**sharedFontPanel**

Returns an NSFontPanel object.

– (NSFont *)**panelConvertFont:**(NSFont *)*fontObject*

Returns a Font object whose traits are the same as those of *fontObject* except as specified by the user's choices in the Font Panel.

Setting the Font

– (void)**setPanelFont:**(NSFont *)*fontObject* **isMultiple:**(BOOL)*flag*

Sets the FontPanel's current font from *fontObject* with *flag* indicating whether it contains multiple fonts.

Configuring the NSFontPanel

– (NSView *)**accessoryView** — Returns the application-customized view.

– (BOOL)**isEnabled** — Returns whether the FontPanel's Set button is enabled.

– (void)**setAccessoryView:**(NSView *)*aView* — Adds *aView* above the action buttons at the bottom of the panel.

– (void)**setEnabled:**(BOOL)*flag* — Enables or disables the FontPanel's Set button depending on *flag*.

– (BOOL)**worksWhenModal** — Returns whether FontPanel works when another window is modal.

Displaying the NSFontPanel

– (void)**orderWindow:**(NSWindowOrderingMode)*place*
relativeTo:(int)*otherWindows* — Repositions the FontPanel above or below the other windows *otherWindows* as indicated by *place* and updates the FontPanel if necessary.

NSForm

Inherits From: NSMatrix : NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSForm.h

Class Description

An NSForm is an NSMatrix that contains titled entries (text fields) into which a user can type data values. Entries are indexed from the top down (starting with zero). Each item in the NSForm, including the titles, is an NSFormCell. A mouse click on an NSFormCell (that is, on the title or in the entry area) starts text editing in that entry. If the user presses the Return or Enter key while editing an entry, the action of the entry is sent to the target of the entry, or—if the entry doesn't have an action—the NSForm sends its action to its target. If the user presses the Tab key, the next entry in the NSForm is selected; if the user presses Shift-Tab, the previous entry is selected.

For more information, see the NSFormCell and NSMatrix class specifications.

Laying Out the Form

– (NSFormCell *)**addEntry:**(NSString *)*title*	Adds and returns a new entry with *title* as its title at the end of the Form.
– (NSFormCell *)**insertEntry:**(NSString *)*title* **atIndex:**(int)*index*	Inserts a new entry at *index* with *title* as its title.
– (void)**removeEntryAtIndex:**(int)*index*	Removes the entry at *index*.
– (void)**setInterlineSpacing:**(float)*spacing*	Sets the spacing between entries to *spacing*.

Finding Indices

– (int)**indexOfCellWithTag:**(int)*aTag*	Returns the index for the entry with tag *aTag*.
– (int)**indexOfSelectedItem**	Returns the index of the currently selected entry.

Modifying Graphic Attributes

– (void)**setBezeled:**(BOOL)*flag*	Sets whether entries have a bezeled border.
– (void)**setBordered:**(BOOL)*flag*	Sets whether the entries have a plain border.
– (void)**setTextAlignment:**(int)*mode*	Sets how text is aligned within the entries to *mode*.
– (void)**setTextFont:**(NSFont *)*fontObject*	Sets the font used to draw entry text to *fontObject*.

– (void)**setTitleAlignment:**(NSTextAlignment)*mode* — Sets how titles are aligned to *mode*.

– (void)**setTitleFont:**(NSFont *)*fontObject* — Sets the font used to draw entry titles to *fontObject*.

Setting the Cell Class

+ (Class)**cellClass** — Returns the class last set in a **setCellClass:** message, or the NSFormCell class if **setCellClass:** has never been called.

+ (void)**setCellClass:**(Class)*classId* — Sets the class of NSCell used in the NSForm.

Getting a Cell

– (id)**cellAtIndex:**(int)*index* — Returns the Cell at *index*.

Displaying a Cell

– (void)**drawCellAtIndex:**(int)*index* — Displays the Cell at the specified *index*.

Editing Text

– (void)**selectTextAtIndex:**(int)*index* — Selects the text in the entry at *index*.

Resizing the Form

– (void)**setEntryWidth:**(float)*width* — Sets the width of all the entries (including the title part) to *width*.

NSFormCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSFormCell.h

Class Description

This class is used to implement entries in an NSForm. It displays a title within itself, on the left-hand side of the cell. Editing is allowed only in the remaining (right-hand) portion.

See the NSForm class specification for more on the use of NSFormCell.

Initializing an NSFormCell

– (id)**initTextCell:**(NSString *)*aString*	Initializes a new NSFormCell with *aString* as its title.

Determining an NSFormCell's Size

– (NSSize)**cellSizeForBounds:**(NSRect)*aRect*	Calculates the NSFormCell's size within *aRect*.

Determining Graphic Attributes

– (BOOL)**isOpaque**	Returns whether the NSFormCell is opaque.

Modifying the Title

– (void)**setTitle:**(NSString *)*aString*	Sets the NSFormCell's title to *aString*.
– (void)**setTitleAlignment:**(NSTextAlignment)*mode*	Sets the alignment of the title to *mode*.
– (void)**setTitleFont:**(NSFont *)*fontObject*	Sets the font used to draw the title to *fontObject*.
– (void)**setTitleWidth:**(float)*width*	Sets the width of the NSFormCell's title field to *width*.
– (NSString *)**title**	Returns the NSFormCell's title.
– (NSTextAlignment)**titleAlignment**	Returns the alignment of the title.
– (NSFont *)**titleFont**	Returns the font used to draw the title.

– (float)**titleWidth** Returns the width of the title.

– (float)**titleWidth:**(NSSize)*aSize* Returns the width of the title, constrained to *aSize*.

Displaying

– (void)**drawInteriorWithFrame:**(NSRect)*cellFrame*
inView:(NSView *)*controlView* Draws only the editable text portion of the FormCell.

NSHelpPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSHelpPanel.h

Class Description

The NSHelpPanel class is the central component of the OpenStep help system. It provides the Help panel that displays the text and illustrations that constitute your application's help information. The NSHelpPanel class object itself stores the table of associations between an application's user-interface objects and specific passages of the help text.

Users can display the Help panel by choosing the Help command from an application's Info menu. The panel employs the metaphor of a book: It displays a table of contents, body text, and an index. Users can browse through the text by clicking entries in the table of contents or index. The panel also supports hypertext-like help links, which appear as diamond-shaped images within the text and allow the user to easily follow cross references. By using the help cursor and clicking user-interface objects, the user can query the Help panel for information associated with those objects.

The Help Text

An NSHelpPanel object looks in a language-specific directory within the application's file package for the text that it will display. (Some implementations may employ more efficient means of storage than files and directories.) For example, if the user's language preference is English, the panel searches for a directory named **Help** within the **English.lproj** directory of the application's file package. It searches for two files: **TableOfContents.rtf** and **Index.rtfd**. There may also be one or more files containing the body text that the Help panel will display. The table-of-contents, index, and body files are interconnected by a system of *help links* and *help markers*.

A help marker is a named position holder in the stream of text—in most cases, it's invisible to users. A help link is a diamond-shaped button embedded in the text. Help links store a file name and, optionally, a help marker name. When a user clicks a help link, the Help panel displays the named file. If the help link also stores a marker name, the displayed file is scrolled to the position of the marker, and the text is selected from the marker's position to the end of the line.

Table-of-Contents and Index Files

The table-of-contents and index files are specially designed documents in Rich Text Format (RTF). An NSHelpPanel object identifies these files by name (**TableOfContents.rtf** and **Index.rtfd**) and processes them differently than it does other help files.

The table-of-contents file should contain one entry for each help text file in the help directory. Each entry begins with a help link that stores the name of the destination file for that entry. Following the link is the text of the entry, which may wrap and span several lines. Although the table of contents in the Help panel looks like it's displayed by an NSMatrix, it's actually displayed by a modified NSText object. Thus, you can use the full generality of RTF to format your table of contents.

The index file is structured similarly although there is no enforced one-to-one mapping. Generally, the help link that begins an index entry stores both a file name and a marker name, since an index entry usually points to a specific word or phrase within a file.

Generic Help Files

An application's Help directory can contain only table-of-contents and index files, and yet the application may be able to display numerous help subjects, each of a general nature. This is because OpenStep applications have access to generic help files contained in a directory found in a system-specific location.

When a help link is being resolved, the NSHelpPanel first looks for the specified file within the appropriate *language***.lproj/Help** directory of the application's file package. If the file isn't found, it then searches the directory of generic help files. This search path is used for all links, whether they are in the table of contents, index, or body text.

If one of these generic help files is inappropriate for your application, you have two remedies: You can remove the table-of-contents and index entries that refer to it, or you can override the file with one that's more appropriate. By placing a file of the same name and relative location within your application's **Help** directory, NSHelpPanel will display it rather than the generic file.

Associating Help Text with Objects

The NSHelpPanel class stores associations between user-interface objects and help text. When the user presses the Help modifier key (which varies depending on the hardware running the application), a question mark cursor appears. If the user clicks an object using this cursor, the Help panel displays the associated help text.

You can attach a help file to a user-interface object programmatically, by sending an **attachHelpFile:markerName:to:** message to the NSHelpPanel class object. This method takes a file name, a marker name, and an object **id** as its arguments. The **detachHelpFrom:** message removes such an association.

Just as with help links, an NSHelpPanel searches both the application's file package and the generic help files in attempting to find the file associated with a particular user-interface object.

Hidden Files

Although in general there's a one-to-one relationship between table-of-contents entries and files in the **Help** directory, you can force a single table-of-contents entry to represent multiple "hidden" files. This can be useful in reducing the overall length of the table of contents.

Hidden files can't be accessed from the table of contents; rather, the user must find them by Help-clicking an object in the application's user interface, by using the Help Panel's Find command, by using the index, or by following a help link from some other file. However, when a hidden file is displayed, the Help panel must select some entry in the table of contents.

Conversely, when the user selects such a table-of-contents entry, the Help panel must display one of the files in the directory of hidden files; by convention, this file must be named **Prolog.rtfd**. The prolog file typically informs users that they can get help on a particular user-interface object by Help-clicking that object.

The Help panel's Find button searches through all the files that are connected to table-of-contents entries, first looking in the application's **Help** directory and then in the generic help material. If you don't want some hidden file in the generic help material to appear in your application's Help panel as the result of a Find operation, override the file with an empty file of the same name. Since the file is empty, no search string will ever be found in it, and it will effectively block the generic file of the same name from being searched.

Searching the Help Text

By clicking the Help panel's Find button, users can search the help text for strings. NSHelpPanel uses two approaches to locate text containing a specific string. First, it attempts to find the string in the currently displayed help text by sending the object that displays the text (an instance of NSCStringText) a **findText:ignoreCase:backwards:wrap:** message. If the search is unsuccessful, or if the search is continued past the last occurrence of the string in the current file, the NSHelpPanel object scans for the string in other help files, both within the application's help files and within the generic help files. Some implementations of NSHelpPanel may make use of a previously built index of all the help text to speed this search.

Help Supplements

Since in OpenStep an application may load executable modules dynamically (for example, a drawing program could allow the user to load a new drawing tool), an NSHelpPanel object provides the ability to load supplemental help information. When the application loads the module, it sends the NSHelpPanel object an **addSupplement:inPath:** message to inform the object of the location of the new help supplement. The NSHelpPanel object appends the contents of the supplement's **TableOfContents.rtf** to the existing table of contents, so the supplement should have a title that clearly sets it off from the main part of the table of contents, for example:

—Pattern Tool Supplement—

- **Pattern Options**
  - `Brick`
  - `Stucco`
  - `W ood`
  - `Tile`
  - `Custom`
- **Resizing and Rotating**
- **Blending Patterns**
- **Index to Supplement**

The supplement's index is only accessible from the table of contents; the Help panel's Index button displays only the main index.

Accessing the Help Panel

+ (NSHelpPanel *)**sharedHelpPanel** — Creates, if necessary, and returns the NSHelpPanel object.

+ (NSHelpPanel *)**sharedHelpPanelWithDirectory:**(NSString *)*helpDirectory* — Creates, if necessary, and returns the NSHelpPanel object. If the panel is created, it loads the help directory specified by *helpDirectory*. The help directory must reside in the main bundle. If a Help panel already exists but has loaded a help directory other than *helpDirectory*, a second panel will be created.

Managing the Contents

+ (void)**setHelpDirectory**:(NSString *)*helpDirectory* — Initializes the panel to display the help text found in *helpDirectory*. By default, the receiver looks for a directory named “Help”.

– (void)**addSupplement:**(NSString *)*helpDirectory* **inPath:**(NSString *)*supplementPath* — Append additional help entries to the Help panel’s table of contents.

– (NSString *)**helpDirectory** — Returns the absolute path of the help directory.

– (NSString *)**helpFile** — Returns the path of the currently loaded help file.

Attaching Help to Objects

+ (void)**attachHelpFile:**(NSString *)*filename* **markerName:**(NSString *)*markerName* **to:**(id)*anObject* — Associates the help file *filename* and *markerName* with *anObject*.

+ (void)**detachHelpFrom:**(id)*anObject* — Removes any help information associated with *anObject*.

Showing Help

– (void)**showFile:**(NSString *)*filename* **atMarker:**(NSString *)*markerName* — Causes the panel to display the help contained in *filename* at *markerName*.

– (BOOL)**showHelpAttachedTo:**(id)*anObject* — Causes the panel to display help attached to *anObject*.

Printing

– (void)**print:**(id)*sender* — Prints the currently displayed help text.

NSImage

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSImage.h

Class Description

An NSImage object contains an image that can be composited anywhere without first being drawn in any particular view. It manages the image by:

- Reading image data from the application bundle, from an NSPasteboard, or from an NSData object.
- Keeping multiple representations of the same image.
- Choosing the representation that's appropriate for a particular data type.
- Choosing the representation that's appropriate for any given display device.
- Caching the representations it uses by rendering them in off-screen windows.
- Optionally retaining the data used to draw the representations, so that they can be reproduced when needed.
- Compositing the image from the off-screen cache to where it's needed on-screen.
- Reproducing the image for the printer so that it matches what's displayed on-screen, yet is the best representation possible for the printed page.
- Automatically using any filtering services installed by the user to convert image data from unsupported formats to supported formats.

Defining an Image

An image can be created from various types of data:

- Encapsulated PostScript code (EPS)
- Bitmap data in Tag Image File Format (TIFF)
- Untagged (raw) bitmap data
- Other image data supported by an NSImageRep subclass registered with the NSImage class
- Data that can be filtered to a supported type by a user-installed filter service

If data is placed in a file (for example, in an application bundle), the NSImage object can access the data whenever it's needed to create the image. If data is read from an NSData object, the NSImage object may need to store the data itself.

Images can also be defined by the program, in two ways:

- By drawing the image in an off-screen window maintained by the NSImage object. In this case, the NSImage maintains only the cached image.
- By defining a method that can be used to draw the image when needed. This allows the NSImage to delegate responsibility for producing the image to some other object.

Image Representations

An NSImage object can keep more than one representation of an image. Multiple representations permit the image to be customized for the display device. For example, different hand-tuned TIFF images can be provided for monochrome and color screens, and an EPS representation or a custom method might be used for printing. All representations are versions of the same image.

An NSImage returns an NSArray of its representations in response to a **representations** message. Each representation is a kind of NSImageRep object:

NSEPSImageRep	An image that can be recreated from EPS data that's either stored by the object or at a known location in the file system.
NSBitmapImageRep	An image that can be recreated from bitmap or TIFF data.
NSCustomImageRep	An image that can be redrawn by a method defined in the application.
NSCachedImageRep	An image that has been rendered in an off-screen cache from data or instructions that are no longer available. The image in the cache provides the only data from which the image can be reproduced.

You can define other NSImageRep subclasses for objects that render images from other types of source data. To make these new subclasses available to an NSImage object, they need to be added to the NSImageRep class registry by invoking the **registerImageRepClass:** class method. NSImage determines the data types that each subclass can support by invoking its **imageUnfilteredFileTypes** and **imageUnfilteredPasteboardTypes** methods.

Choosing Representations

The NSImage object will choose the representation that best matches the rendering device. By default, the choice is made according to the following set of ordered rules. Each rule is applied in turn until the choice of representation is narrowed to one.

1. Choose a color representation for a color device, and a gray-scale representation for a monochrome device.
2. Choose a representation with a resolution that matches the resolution of the device, or if no representation matches, choose the one with the highest resolution.

By default, any image representation with a resolution that's an integer multiple of the device resolution is considered to match. If more than one representation matches, the NSImage will choose the one that's closest to the device resolution. However, you can force resolution matches to be exact by passing NO to the **setMatchesOnMultipleResolution:** method.

Rule 2 prefers TIFF and bitmap representations, which have a defined resolution, over EPS representations, which don't. However, you can use the **setUsesEPSOnResolutionMismatch:** method to have the NSImage choose an EPS representation in case a resolution match isn't possible.

3. If all else fails, choose the representation with a specified bits per sample that matches the depth of the device. If no representation matches, choose the one with the highest bits per sample.

By passing NO to the **setPrefersColorMatch:** method, you can have the NSImage try for a resolution match before a color match. This essentially inverts the first and second rules above.

If these rules fail to narrow the choice to a single representation—for example, if the NSImage has two color TIFF representations with the same resolution and depth—the one that will be chosen is system dependent.

Caching Representations

When first asked to composite the image, the NSImage object chooses the representation that's best for the destination display device, as outlined above. It renders the representation in an off-screen window on the same device, then composites it from this cache to the desired location. Subsequent requests to composite the image use the same cache. Representations aren't cached until they're needed for compositing.

When printing, the NSImage tries not to use the cached image. Instead, it attempts to render on the printer—using the appropriate image data, or a delegated method—the best version of the image that it can. Only as a last resort will it image the cached bitmap.

Image Size

Before an NSImage can be used, the size of the image must be set, in units of the base coordinate system. If a representation is smaller or larger than the specified size, it can be scaled to fit.

If the size of the image hasn't already been set when the NSImage is provided with a representation, the size will be set from the data. The bounding box is used to determine the size of an NSEPSImageRep. The TIFF fields "ImageLength" and "ImageWidth" are used to determine the size of an NSBitmapImageRep.

Coordinate Systems

Images have the horizontal and vertical orientation of the base coordinate system; they can't be rotated or flipped. When composited, an image maintains this orientation, no matter what coordinate system it's composited to. (The destination coordinate system is used only to determine the location of a composited image, not its size or orientation.)

It's possible to refer to portions of an image when compositing by specifying a rectangle in the image's coordinate system, which is identical to the base coordinate system, except that the origin is at the lower left corner of the image.

Named Images

An NSImage object can be identified either by its **id** or by a name. Assigning an NSImage a name adds it to a table kept by the class object; each name in the database identifies one and only one instance of the class. When you ask for an NSImage object by name (with the **imageNamed:** method), the class object returns the one from its database, which also includes all the system bitmaps provided by the Application Kit. If there's no object in the database for the specified name, the class object tries to create one by checking for a system bitmap of the same name, checking the name of the application's own image, and then checking for the image in the application's main bundle.

If a section or file matches the name, an NSImage is created from the data stored there. You can therefore create NSImage objects simply by including EPS or TIFF data for them within the executable file, or in files inside the application's file package.

Image Filtering Services

NSImage is designed to automatically take advantage of user-installed filter services for converting unsupported image file types to supported image file types. The class method **imageFileTypes** returns an array of all file types from which NSImage can create an instance of itself. This list includes all file types supported by registered subclasses of NSImageRep, and those types that can be converted to supported file types through a user-installed filter service.

Initializing a New NSImage Instance

– (id)**initByReferencingFile:**(NSString *)*filename* — Initializes the new NSImage from the data in *filename*. The file is assumed to persist and may be reread later if the NSImage is resized or otherwise modified.

– (id)**initWithContentsOfFile:**(NSString *)*filename* — Initializes the new NSImage from the data in *filename*.

– (id)**initWithData:**(NSData *)*data* — Initializes the new NSImage from *data*.

– (id)**initWithPasteboard:**(NSPasteboard *)*pasteboard* — Initializes the new NSImage with the data in *pasteboard*.

– (id)**initWithSize:**(NSSize)*aSize* — Initializes the new NSImage to the specified size.

Setting the Size of the Image

– (void)**setSize:**(NSSize)*aSize* — Sets the size of the image to *aSize* in base coordinates.

– (NSSize)**size** — Returns the size of the image.

Referring to Images by Name

+ (id)**imageNamed:**(NSString *)*name* — Returns the NSImage object having *name*. Searches the main bundle for the image if necessary.

– (BOOL)**setName:**(NSString *)*name*
Assigns *name* to be the receiver's name. Returns NO if *name* is already in use; otherwise, returns YES.

– (NSString *)**name**
Returns the receiver's name.

Specifying the Image

– (void)**addRepresentation:**(NSImageRep *)*imageRep*
Adds *imageRep* to the receiver's list of representations.

– (void)**addRepresentations:**(NSArray *)*imageRepArray*
Adds the *imageRep*s from *imageRepArray* to the receiver's list of representations.

– (void)**lockFocus**
Prepares for drawing in the best representation.

– (void)**lockFocusOnRepresentation:**(NSImageRep *)*imageRep*
Prepares for drawing in *imageRep*.

– (void)**unlockFocus**
Balances a previous **lockFocus** or **lockFocusOnRepresentation:**.

Using the Image

– (void)**compositeToPoint:**(NSPoint)*aPoint* **operation:**(NSCompositingOperation)*op*
Composites the image to *aPoint* using the operation *op*.

– (void)**compositeToPoint:**(NSPoint)*aPoint* **fromRect:**(NSRect)*aRect* **operation:**(NSCompositingOperation)*op*
Composites the *aRect* portion of the image to *aPoint* using the operation *op*.

– (void)**dissolveToPoint:**(NSPoint)*aPoint* **fraction:**(float)*aFloat*
Composites the image to *aPoint* using the **dissolve** operator. *aFloat* is a value from 0.0 to 1.0 that determines how much of the resulting composite comes from the receiver.

– (void)**dissolveToPoint:**(NSPoint)*aPoint* **fromRect:**(NSRect)*aRect* **fraction:**(float)*aFloat*
Composites the *aRect* portion of the image to *aPoint* using the **dissolve** operator. *aFloat* is a value from 0.0 to 1.0 that determines how much of the resulting composite comes from the receiver.

Choosing Which Image Representation to Use

– (void)**setPrefersColorMatch:**(BOOL)*flag*
Determines whether color matches are preferred.

– (BOOL)**prefersColorMatch**
Returns whether color matches are preferred.

– (void)**setUsesEPSOnResolutionMismatch:**(BOOL)*flag*
Sets whether to use EPS representations on mismatch.

– (BOOL)**usesEPSOnResolutionMismatch** Returns whether to use EPS representations on mismatch.

– (void)**setMatchesOnMultipleResolution:**(BOOL)*flag*
Sets whether resolution multiples match.

– (BOOL)**matchesOnMultipleResolution** Returns whether resolution multiples match.

Getting the Representations

– (NSImageRep *)**bestRepresentationForDevice:**(NSDictionary *)*deviceDescription*
Returns the best representation for the device described by *deviceDescription*. If *deviceDescription* is **nil**, the current device is assumed. See **NSGraphics.h** for appropriate dictionary keys and values.

– (NSArray *)**representations** Returns an array of all the representations.

– (void)**removeRepresentation:**(NSImageRep *)*imageRep*
Removes *imageRep* from the receiver's list of representations.

Determining How the Image is Stored

– (void)**setCachedSeparately:**(BOOL)*flag* Sets whether representations are cached separately.

– (BOOL)**isCachedSeparately** Returns whether representations are cached separately.

– (void)**setDataRetained:**(BOOL)*flag* Sets whether image data is retained by the object after the image is cached.

– (BOOL)**isDataRetained** Returns whether image data is retained.

– (void)**setCacheDepthMatchesImageDepth:**(BOOL)*flag*
Sets whether the default depth limit applies to caches.

– (BOOL)**cacheDepthMatchesImageDepth** Returns whether the default depth limit applies to caches.

Determining How the Image is Drawn

– (BOOL)**isValid** Returns YES to indicate that the receiver's image is valid.

– (void)**setScalesWhenResized:**(BOOL)*flag* If flag is YES, representations are scaled to fit.

– (BOOL)**scalesWhenResized** Returns whether representations are scaled to fit.

– (void)**setBackgroundColor:**(NSColor *)*aColor* Sets the background color of the image to *aColor.*

– (NSColor *)**backgroundColor** Returns the background color of the image.

– (BOOL)**drawRepresentation:**(NSImageRep *)*imageRep* **inRect:**(NSRect)*aRect*
Overridden to have *imageRep* draw the representation in *aRect*.

– (void)**recache**
Invalidates caches of all representations, so they will be redrawn.

Assigning a Delegate

– (void)**setDelegate:**(id)*anObject*
Makes *anObject* the delegate of the NSImage.

– (id)**delegate**
Returns the delegate of the NSImage.

Producing TIFF Data for the Image

– (NSData *)**TIFFRepresentation**
Returns a data object containing TIFF for all representations, using their default compressions.

– (NSData *)**TIFFRepresentationUsingCompression:**(NSTIFFCompression)*comp* **factor:**(float)*aFloat*
Returns a data object containing TIFF for all the representations.

Managing NSImageRep Subclasses

+ (NSArray *)**imageUnfilteredFileTypes**
Returns an array of file types recognized by the NSImage without filtering. This list comes from all registered NSImageReps.

+ (NSArray *)**imageUnfilteredPasteboardTypes**
Returns an array of pasteboard types recognized by the NSImage.

Testing Image Data Sources

+ (BOOL)**canInitWithPasteboard:**(NSPasteboard *)*pasteboard*
Returns YES if the receiver can create a representation from *pasteboard*; otherwise, returns NO.

+ (NSArray *)**imageFileTypes**
Returns an array of supported image data file types.

+ (NSArray *)**imagePasteboardTypes**
Returns an array of supported pasteboard types.

Methods Implemented by the Delegate

– (NSImage *)**imageDidNotDraw:**(id)*sender* **inRect:**(NSRect)*aRect*
Responds to message that *image* couldn’t be composited into *aRect*.

NSImageRep

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSImageRep.h

Class Description

NSImageRep is an abstract superclass; each of its subclasses knows how to draw an image from a particular kind of source data. While an NSImageRep subclass can be used directly, it's typically used through an NSImage object. An NSImage manages a group of representations, choosing the best one for the current output device.

There are four subclasses defined in the Application Kit:

Subclass	Source Data
NSBitmapImageRep	Tag Image File Format (TIFF) and other bitmap data
NSEPSImageRep	Encapsulated PostScript code (EPS)
NSCustomImageRep	A delegated method that can draw the image
NSCachedImageRep	A rendered image, usually in an off-screen window

You can define other NSImageRep subclasses for objects that render images from other types of source information. New subclasses must be added to the NSImageRep class registry by invoking the **registerImageRepClass:** class method. The NSImageRep subclass informs the registry of the data types it can support through its **imageUnfilteredFileTypes**, **imageUnfilteredPasteboardTypes**, and **canInitWithData:** class methods. Once an NSImageRep subclass is registered, an instance of that subclass is created anytime NSImage encounters the type of data handled by that subclass.

Creating an NSImageRep

+ (id)**imageRepWithContentsOfFile:**(NSString *)*filename*

In subclasses that respond to **imageFileTypes** and **imageRepWithData:**, returns an object that has been initialized with the data in *filename*. NSImageRep's implementation returns an instance of the appropriate registered subclass.

+ (NSArray *)**imageRepsWithContentsOfFile:**(NSString *)*filename*

In subclasses that respond to **imageFileTypes** and **imageRepWithData:** (or **imageRepWithData:**), returns an array of objects that have been initialized with the data in *filename*. NSImageRep's implementation returns an array of objects (each an instance of the appropriate registered subclass) that have been initialized with the data in *filename*.

+ (id)**imageRepWithPasteboard:**(NSPasteboard *)*pasteboard*

In subclasses that respond to **imagePasteboardTypes** and **imageRepWithData:**, returns an object that has been initialized with the data in *pasteboard*. NSImageRep's implementation returns an instance of the appropriate registered subclass.

+ (NSArray *)**imageRepsWithPasteboard:**(NSPasteboard *)*pasteboard*

In subclasses that respond to **imagePasteboardTypes** and **imageRepsWithData:** (or **imageRepWithData:**), returns an array of objects that have been initialized with the data in *pasteboard*. NSImageRep's implementation returns an array of objects (each an instance of the appropriate registered subclass) that have been initialized with the data in *pasteboard*.

Checking Data Types

+ (BOOL)**canInitWithData:**(NSData *)*data*

Overridden in subclasses to return YES if the receiver can initialize itself from *data*.

+ (BOOL)**canInitWithPasteboard:**(NSPasteboard *)*pasteboard*

Overridden in subclasses to returnYES if the receiver can initialize itself from *pasteboard*.

+ (NSArray *)**imageFileTypes**

Returns an array of strings representing all file types.

+ (NSArray *)**imagePasteboardTypes**

Returns an array of strings representing all pasteboard types.

+ (NSArray *)**imageUnfilteredFileTypes**

Returns an array of strings representing directly supported file types.

+ (NSArray *)**imageUnfilteredPasteboardTypes**

Returns an array of strings representing directly supported pasteboards.

Setting the Size of the Image

– (void)**setSize:**(NSSize)*aSize*	Sets the size of the image.
– (NSSize)**size**	Returns the size of the image.

Specifying Information about the Representation

– (int)**bitsPerSample**	Returns the number of bits per pixel in each component.
– (NSString *)**colorSpaceName**	Returns the name of the image's color space.
– (BOOL)**hasAlpha**	Returns whether there is a coverage component.
– (BOOL)**isOpaque**	Returns whether the representation is opaque.
– (int)**pixelsHigh**	Returns the height specified in the image data.
– (int)**pixelsWide**	Returns the width specified in the image data.
– (void)**setAlpha:**(BOOL)*flag*	Informs the receiver whether there is a coverage component.
– (void)**setBitsPerSample:**(int)*anInt*	Informs the receiver there are *anInt* bits/pixel in a component.
– (void)**setColorSpaceName:**(NSString *)*aString*	Informs the receiver of the image's color space.
– (void)**setOpaque:**(BOOL)*flag*	Informs the receiver of the image's opacity.
– (void)**setPixelsHigh:**(int)*anInt*	Informs the receiver that its data is for an image *anInt* pixels high.
– (void)**setPixelsWide:**(int)*anInt*	Informs the receiver that its data is for an image *anInt* pixels wide.

Drawing the Image

– (BOOL)**draw**	Implemented by subclasses to draw the image.
– (BOOL)**drawAtPoint:**(NSPoint)*aPoint*	Modifies current coordinates so the image is drawn at *aPoint*.
– (BOOL)**drawInRect:**(NSRect)*aRect*	Modifies current coordinates so the image is drawn in *aRect*.

Managing NSImageRep Subclasses

+ (Class)**imageRepClassForData:**(NSData *)*data*
Returns the NSImageRep subclass that handles data of type *data*.

+ (Class)**imageRepClassForFileType:**(NSString *)*type*
Returns the NSImageRep subclass that handles data of file type *type*.

+ (Class)**imageRepClassForPasteboardType:**(NSString *)*type*
Returns the NSImageRep subclass that handles data of pasteboard type *type*.

+ (void)**registerImageRepClass:**(Class)*imageRepClass*
Adds *imageRepClass* to the registry of available NSImageRep classes. This method posts the NSImageRepRegistryChangedNotification notification with the receiving object to the default notification center.

+ (NSArray *)**registeredImageRepClasses**
Returns the names of the registered NSImageRep classes.

+ (void)**unregisterImageRepClass:**(Class)*imageRepClass*
Removes *imageRepClass* from the registry of available NSImageRep classes. This method posts the NSImageRepRegistryChangedNotification notification with the receiving object to the default notification center.

NSMatrix

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSMatrix.h

Class Description

NSMatrix is a class used for creating groups of NSCells that work together in various ways. It includes methods for arranging NSCells in rows and columns, either with or without space between them. NSCells in an NSMatrix are numbered by row and column, each starting with 0; for example, the top left NSCell would be at (0, 0), and the NSCell that's second down and third across would be at (1, 2).

The cell objects that an NSMatrix contains are usually of a single subclass of NSCell, but they can be of multiple subclasses of NSCell. The only restriction is that all cell objects must be the same size. An NSMatrix can be set up to create new NSCells by copying a prototype object, or by allocating and initializing instances of a specific NSCell class.

An NSMatrix adds to NSControl's target/action paradigm by allowing a separate target and action for each of its NSCells in addition to its own target and action. It also allows for an action message that's sent when the user double-clicks an NSCell, and which is sent in addition to the single-click action message. If an NSCell doesn't have an action, the NSMatrix sends its own action to its own target. If an NSCell doesn't have a target, the NSMatrix sends the NSCell's action to its own target. The double-click action of an NSMatrix is always sent to the target of the NSMatrix.

Since the user might press the mouse button while the cursor is within the NSMatrix and then drag the mouse around, NSMatrix offers four "selection modes" that determine how NSCells behave when the NSMatrix is tracking the mouse:

- NSTrackModeMatrix is the most basic mode of operation. In this mode the NSCells are asked to track the mouse with **trackMouse:inRect:ofView:untilMouseUp:** whenever the mouse is inside their bounds. No highlighting is performed. An example of this mode might be a "graphic equalizer" NSMatrix of sliders, where moving the mouse around causes the sliders to move under the mouse.
- NSHighlightModeMatrix is a modification of NSTrackModeMatrix. In this mode, an NSCell is highlighted before it's asked to track the mouse, then unhighlighted when it's done tracking. This is useful for multiple unconnected NSCells that use highlighting to inform the user that they are being tracked (like push-buttons and switches).
- NSRadioModeMatrix is used when you want no more than one NSCell to be selected at a time. It can be used to create a set of buttons of which one and only one is selected (there's the option of allowing no button to be selected). Any time an NSCell is selected, the previously selected NSCell is unselected. The canonical example of this mode is a set of radio buttons.

- NSListModeMatrix is the opposite of NSTrackModeMatrix. NSCells are highlighted, but don't track the mouse. This mode can be used to select a range of text values, for example. NSMatrix supports the standard multiple-selection paradigms of dragging to select, using the shift key to make discontinuous selections, and using the alternate key to extend selections.

Initializing the NSMatrix Class

+ (Class)**cellClass** — Returns the default class used to make cells.

+ (void)**setCellClass:**(Class)*classId* — Sets the default class used to make cells.

Initializing an NSMatrix Object

– (id)**initWithFrame:**(NSRect)*frameRect* — Initializes a new NSMatrix object in *frameRect*.

– (id)**initWithFrame:**(NSRect)*frameRect*
mode:(int)*aMode*
cellClass:(Class)*classId*
numberOfRows:(int)*rowsHigh*
numberOfColumns:(int)*colsWide*
— Initializes a new NSMatrix object in *frameRect*, with *aMode* as the selection mode, *classId* as the class used to make new cells, and having *rowsHigh* rows and *colsWide* columns.

– (id)**initWithFrame:**(NSRect)*frameRect*
mode:(int)*aMode*
prototype:(NSCell *)*aCell*
numberOfRows:(int)*rowsHigh*
numberOfColumns:(int)*colsWide*
— Initializes a new NSMatrix object with the given values with *aMode* as the selection mode, *aCell* as the prototype copied to make new cells, and having *rowsHigh* rows and *colsWide* columns.

Setting the Selection Mode

– (NSMatrixMode)**mode** — Returns the selection mode of the matrix.

– (void)**setMode:**(NSMatrixMode)*aMode* — Sets the selection mode of the matrix.

Configuring the NSMatrix

– (BOOL)**allowsEmptySelection** — Returns whether it's possible to have no cells selected.

– (BOOL)**isSelectionByRect** — Returns whether a user can drag a rectangular selection.

– (void)**setAllowsEmptySelection:**(BOOL)*flag* — Sets whether it's possible to have no cells selected.

– (void)**setSelectionByRect:**(BOOL)*flag* — Sets whether a user can drag a rectangular selection (the default is YES). If *flag* is NO, selection is on a row-by-row basis.

Setting the Cell Class

– (Class)**cellClass**	Returns the subclass of NSCell used to make new cells.
– (id)**prototype**	Returns the prototype cell copied to make new cells.
– (void)**setCellClass:**(Class)*classId*	Sets the subclass of NSCell used to make new cells.
– (void)**setPrototype:**(NSCell *)*aCell*	Sets the prototype cell copied to make new cells.

Laying Out the NSMatrix

– (void)**addColumn**	Adds a new column of cells to the right of the last column.
– (void)**addColumnWithCells:**(NSArray *)*cellArray*	Adds a new column of cells, using those contained in *cellArray.*
– (void)**addRow**	Adds a new row of cells below the last row.
– (void)**addRowWithCells:**(NSArray *)*cellArray*	Adds a new row of cells, using those contained in *cellArray.*
– (NSRect)**cellFrameAtRow:**(int)*row* **column:**(int)*column*	Returns the frame rectangle of the cell at *row* and *column*.
– (NSSize)**cellSize**	Returns the width and height of cells in the matrix.
– (void)**getNumberOfRows:**(int *)*rowCount* **columns:**(int *)*columnCount*	Gets the number of rows and columns in the matrix.
– (void)**insertColumn:**(int)*column*	Inserts a new column of cells at *column*, creating as many as needed to make the matrix *column* columns wide.
– (void)**insertColumn:**(int)*column* **withCells:**(NSArray *)*cellArray*	Inserts a new row of cells at *column*, using those contained in *cellArray.*
– (void)**insertRow:**(int)*row*	Inserts a new row of cells at *row*, creating as many as needed to make the matrix *row* rows wide.
– (void)**insertRow:**(int)*row* **withCells:**(NSArray *)*cellArray*	Inserts a new row of cells at *row*, using those contained in *cellArray.*
– (NSSize)**intercellSpacing**	Returns the vertical and horizontal spacing between cells
– (NSCell *)**makeCellAtRow:**(int)*row* **column:**(int)*column*	Creates a new cell at *row*, *column* in the matrix and returns it.
– (void)**putCell:**(NSCell *)*newCell* **atRow:**(int)*row* **column:**(int)*column*	Replaces the cell at *row* and *column* with *newCell*.

– (void)**removeColumn:**(int)*column* — Removes the column at *column*, releasing the cells.

– (void)**removeRow:**(int)*row* — Removes the row at *row*, releasing the cells.

– (void)**renewRows:**(int)*newRows*
columns:(int)*newColumns* — Changes the number of rows and columns in the receiver without freeing any cells.

– (void)**setCellSize:**(NSSize)*aSize* — Sets the width and height of all cells in the matrix.

– (void)**setIntercellSpacing:**(NSSize)*aSize* — Sets the vertical and horizontal spacing between cells.

– (void)**sortUsingFunction:**(int (*)(id *element1*, id *element2*, void **userData*))*comparator*
context:(void *)*context* — Sorts the receiver's cells in ascending order as defined by the comparison function *comparator. context* is passed as the function's third argument.

– (void)**sortUsingSelector:**(SEL)*comparator* — Sorts the receiver's cells in ascending order as defined by the comparison method *comparator.*

Finding Matrix Coordinates

– (BOOL)**getRow:**(int *)*row*
column:(int *)*column*
forPoint:(NSPoint)*aPoint* — Gets the row and column position corresponding to *aPoint*. Returns YES if *aPoint* is within the matrix; NO otherwise.

– (BOOL)**getRow:**(int *)*row*
column:(int *)*column*
ofCell:(NSCell *)*aCell* — Gets the row and column position of *aCell*. Returns YES if *aCell* is in the matrix; NO otherwise.

Modifying Individual Cells

– (void)**setState:**(int)*value*
atRow:(int)*row*
column:(int)*column* — Sets the state of the cell at *row* and *column* to *value*.

Selecting Cells

– (void)**deselectAllCells** — Clears the receiver's selection, assuming that the NSMatrix allows an empty selection.

– (void)**deselectSelectedCell** — Deselects the selected cell.

– (void)**selectAll:**(id)*sender* — Selects all the cells in the matrix.

– (void)**selectCellAtRow:**(int)*row*
column:(int)*column* — Selects the cell at *row* and *col*.

– (BOOL)**selectCellWithTag:**(int)*anInt* — Selects the cell with the tag *anInt*.

– (id)**selectedCell** Returns the most recently selected cell or **nil** if no cell has been selected.

– (NSArray *)**selectedCells** Returns an array containing the selected cells.

– (int)**selectedColumn** Returns the column of the selected cell or –1 if no column has been selected.

– (int)**selectedRow** Returns the row of the selected cell or –1 if no row has been selected.

– (void)**setSelectionFrom:**(int)*startPos*
to:(int)*endPos*
anchor:(int)*anchorPos*
highlight:(BOOL)*flag*
Selects the cells in the matrix from *startPos* to *endPos*, counting in row order from the upper left, as though *anchorPos* were the number of the last cell selected, and highlighting the cells according to *flag*.

Finding Cells

– (id)**cellAtRow:**(int)*row*
column:(int)*column*
Returns the cell at row *row* and column *col*.

– (id)**cellWithTag:**(int)*anInt* Returns the cell having *anInt* as its tag.

– (NSArray *)**cells** Returns the matrix's array of cells.

Modifying Graphic Attributes

– (NSColor *)**backgroundColor** Returns the color of the background between cells.

– (NSColor *)**cellBackgroundColor** Returns the color of the background within cells.

– (BOOL)**drawsBackground** Returns whether the receiver draws the background between cells.

– (BOOL)**drawsCellBackground** Returns whether the receiver draws the background within cells.

– (void)**setBackgroundColor:**(NSColor *)*aColor* Sets the color of the background between cells to *aColor.*

– (void)**setCellBackgroundColor:**(NSColor *)*aColor*
Sets the color of the background within cells to *aColor.*

– (void)**setDrawsBackground:**(BOOL)*flag* Sets whether the receiver draws the background between cells.

– (void)**setDrawsCellBackground:**(BOOL)*flag* Sets whether the receiver draws the background within cells.

Editing Text in Cells

– (void)**selectText:**(id)*sender*
Selects the text in the first or last editable cell.

– (id)**selectTextAtRow:**(int)*row* **column:**(int)*column*
Selects the text of the cell at *row*, *column* in the matrix.

– (void)**textDidBeginEditing:**(NSNotification *)*notification*
Invoked when there's a change in the text after the receiver gains first responder status. Default behavior is pass to this message on to the text delegate. This method posts the NSControlTextDidBeginEditingNotification notification with the receiving object and, in the notification's dictionary, the text object (with the key NSFieldEditor) to the default notification center.

– (void)**textDidChange:**(NSNotification *)*notification*
Invoked upon a key-down event or paste operation that changes the receiver's contents. Default behavior is to pass this message on to the text delegate. This method posts the NSControlTextDidChangeNotification notification with the receiving object and, in the notification's dictionary, the text object (key NSFieldEditor) to the default notification center.

– (void)**textDidEndEditing:**(NSNotification *)*notification*
Invoked when text editing ends and then forwarded to the text delegate. This method posts the notification NSControlTextDidEndEditingNotification with the receiving object and, in the notification's dictionary, the text object (with the key NSFieldEditor) to the default notification center.

– (BOOL)**textShouldBeginEditing:**(NSText *)*textObject*
Invoked to let the NSTextField respond to impending changes to its text and then forwarded to the text delegate.

– (BOOL)**textShouldEndEditing:**(NSText *)*textObject*
Invoked to let the NSTextField respond to impending loss of first responder status and then forwarded to the text delegate.

Setting Tab Key Behavior

– (id)**nextText**
Returns the object to be selected when the user presses Tab while editing the last text cell.

– (id)**previousText** — Returns the object to be selected when the user presses Shift-Tab while editing the first text cell.

– (void)**setNextText:**(id)*anObject* — Sets the object to be selected when the user presses Tab while editing the last text cell.

– (void)**setPreviousText:**(id)*anObject* — Sets the object to be selected when user presses Shift-Tab while editing the first text cell.

Assigning a Delegate

– (void)**setDelegate:**(id)*anObject* — Sets the delegate for messages from the field editor.

– (id)**delegate** — Returns the delegate for messages from the field editor.

Resizing the Matrix and Cells

– (BOOL)**autosizesCells** — Returns whether the matrix resizes its cells automatically.

– (void)**setAutosizesCells:**(BOOL)*flag* — Sets whether the matrix resizes its cells automatically.

– (void)**setValidateSize:**(BOOL)*flag* — Sets whether the cell size needs to be recalculated.

– (void)**sizeToCells** — Resizes the matrix to fit its cells exactly.

Scrolling

– (BOOL)**isAutoscroll** — Returns whether the matrix automatically scrolls when dragged in.

– (void)**scrollCellToVisibleAtRow:**(int)*row* **column:**(int)*column* — Scrolls the matrix so that the cell at *row* and *column* is visible.

– (void)**setAutoscroll:**(BOOL)*flag* — Sets whether the matrix automatically scrolls when dragged in.

– (void)**setScrollable:**(BOOL)*flag* — If *flag* is YES, makes all the cells scrollable.

Displaying

– (void)**drawCellAtRow:**(int)*row* **column:**(int)*column* — Displays the cell at *row* and *col*.

– (void)**highlightCell:**(BOOL)*flag* **atRow:**(int)*row* **column:**(int)*column* — Highlights (or unhighlights) the cell at *row*, *col*.

Target and Action

– (SEL)**doubleAction**	Returns the action method for double clicks.
– (void)**setDoubleAction:**(SEL)*aSelector*	Sets the action method used on double-clicks to *aSelector.*
– (SEL)**errorAction**	Returns the action method for editing errors.
– (BOOL)**sendAction**	Sends the selected cell's action, or the NSMatrix's action if the cell doesn't have one.
– (void)**sendAction:**(SEL)*aSelector* **to:**(id)*anObject* **forAllCells:**(BOOL)*flag*	Sends *aSelector* to *anObject*, for all cells if *flag* is YES.
– (void)**sendDoubleAction**	Sends the action corresponding to a double-click.
– (void)**setErrorAction:**(SEL)*aSelector*	Sets the action method for editing errors to *aSelector.*

Handling Event and Action Messages

– (BOOL)**acceptsFirstMouse:**(NSEvent *)*theEvent*	Returns NO only if receiver's mode is NSListModeMatrix.
– (void)**mouseDown:**(NSEvent *)*theEvent*	Responds to a mouse-down event. A mouse-down event in a text cell initials editing mode. A double-click in any cell type except a text cell sends the double-click action of the NSMatrix (if there is one) in addition to the single-click action.
– (int)**mouseDownFlags**	Returns the event flags in effect at start of tracking.
– (BOOL)**performKeyEquivalent:**(NSEvent *)*theEvent*	Simulates a mouse click in the appropriate cell.

Managing the Cursor

– (void)**resetCursorRects**	Resets cursor rectangles so that the cursor becomes an I-beam over text cells.

NSMenu

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSMenu.h

Class Description

This class defines an object that manages an application's menus. An NSMenu object displays a list of items that a user can choose from. When an item is clicked, it may either issue a command directly or bring up another menu (a *submenu*) that offers further choices. An NSMenu object's choices are implemented as a column of NSMenuCells in an NSMatrix.

Each NSMenuCell can be configured to send its action message to a target, or to bring up a submenu. When the user clicks a submenu item, the submenu is displayed on the screen, attached to its supermenu so that if the user drags the supermenu, the submenu follows it. A submenu may also be torn away from its supermenu, in which case it displays a close button.

Exactly one NSMenu created by the application is designated as the main menu for the application (with NSApplication's **setMainMenu:** method). This menu is displayed on top of all other windows whenever the application is active, and should never display a close button (because the main menu doesn't have a supermenu).

See the NSMenuCell and NSMatrix class specificiations for more details.

Controlling Allocation Zones

+ (NSZone *)**menuZone** — Returns the zone from which NSMenus should be allocated, creating one if necessary.

+ (void)**setMenuZone:**(NSZone *)*zone* — Sets the zone from which NSMenus should be allocated.

Initializing a New NSMenu

– (id)**initWithTitle:**(NSString *)*aTitle* — Initializes and returns a new NSMenu using *aTitle* for its title.

Setting Up the Menu Commands

– (id)**addItemWithTitle:**(NSString *)*aString*
action:(SEL)*aSelector*
keyEquivalent:(NSString *)*charCode* — Adds a new item with title *aString*, action *aSelector*, and key equivalent *charCode* to the end of the NSMenu. Returns the new NSMenuCell.

– (id)**insertItemWithTitle:**(NSString *)*aString* **action:**(SEL)*aSelector* **keyEquivalent:**(NSString *)*charCode* **atIndex:**(unsigned int)*index*
Adds a new item at *index* having the title *aString*, action *aSelector*, and key equivalent *charCode*. Returns the new NSMenuCell.

– (NSMatrix *)**itemMatrix**
Returns the NSMatrix of NSMenuCell items.

– (void)**setItemMatrix:**(NSMatrix *)*aMatrix*
Replaces the current matrix of items with *aMatrix*.

Finding Menu Items

– (id)**cellWithTag:**(int)*aTag*
Returns the NSMenuCell that has *aTag* as its tag.

Building Submenus

– (NSMenuCell *)**setSubmenu:**(NSMenu *)*aMenu* **forItem:**(NSMenuCell *)*aCell*
Makes *aMenu* a submenu controlled by *aCell*.

– (void)**submenuAction:**(id)*sender*
Activates a submenu attached to sender's NSMenu.

Managing NSMenu Windows

– (NSMenu *)**attachedMenu**
Returns the NSMenu attached to the receiver or **nil** if there's no such object.

– (BOOL)**isAttached**
Returns YES if the receiver is attached to another menu and NO otherwise.

– (BOOL)**isTornOff**
Returns NO if the receiver is attached to another menu (or if it's the main menu) and YES otherwise.

– (NSPoint)**locationForSubmenu:**(NSMenu *)*aSubmenu*
Determines where to display an attached submenu when it's brought up.

– (void)**sizeToFit**
Resizes the receiver to exactly fit the command items.

– (NSMenu *)**supermenu**
Returns the receiver's supermenu.

Displaying the Menu

– (BOOL)**autoenablesItems**
Returns whether the receiver enables and disables its NSMenuCells. (See the NSMenuActionResponder informal protocol.)

– (void)**setAutoenablesItems:**(BOOL)*flag*
Sets whether the receiver enables and disables its NSMenuCells. (See the NSMenuActionResponder informal protocol.)

NSMenuCell

Inherits From: NSButtonCell : NSActionCell : NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSMenuCell.h

Class Description

NSMenuCell is a subclass of NSButtonCell that defines objects that are used in menus. NSMenuCells draw their text left-justified and show an optional key equivalent or submenu arrow on the right. See the NSMenu class specification for more information.

Checking for a Submenu

– (BOOL)**hasSubmenu** — Returns YES if the receiver has a submenu.

Managing User Key Equivalents

+ (void)**setUsesUserKeyEquivalents:**(BOOL)*flag* — If *flag* is YES, NSMenuCells conform to user preferences for key equivalents; otherwise, the key equivalents originally assigned to the NSMenuCells are used.

+ (BOOL)**usesUserKeyEquivalents** — Returns YES if NSMenuCells conform to user preferences for key equivalents; otherwise, returns NO.

– (NSString *)**userKeyEquivalent** — Returns the user-assigned key equivalent for the NSMenuCell.

NSOpenPanel

Inherits From: NSSavePanel : NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSOpenPanel.h

Class Description

NSOpenPanel provides the Open panel of the OpenStep user interface. Applications use the Open panel as a convenient way to query the user for the name of a file to open. The Open panel can only be run modally.

Most of this class's behavior is defined by its superclass, NSSavePanel. NSOpenPanel adds to this behavior by:

- Letting you specify the types (by file-name extension) of the items that will appear in the panel
- Letting the user select files, directories, or both
- Letting the user select multiple items at a time

Typically, you access an NSOpenPanel by invoking the **openPanel** method. When the class receives an **openPanel** message, it tries to reuse an existing panel rather than create a new one. If a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because Open panels may be reused, you shouldn't modify the instance returned by **openPanel**, except through the methods listed below (and those inherited from its superclass, NSSavePanel). For example, you can set the panel's title and whether it allows multiple selection, but not the arrangement of the buttons within the panel. If you must modify the Open panel substantially, create and manage your own instance using the **alloc...** and **init...** methods rather than the **openPanel** method.

Accessing the NSOpenPanel

+ (NSOpenPanel *)**openPanel**	Returns an NSOpenPanel object having default initialization.

Filtering Files

– (BOOL)**allowsMultipleSelection**	Returns YES if the panel allows the user to open multiple files (and directories) at a time.
– (BOOL)**canChooseDirectories**	Returns YES if the panel allows the user to choose directories.
– (BOOL)**canChooseFiles**	Returns YES if the panel allows the user to choose files.

– (void)**setAllowsMultipleSelection:**(BOOL)*flag* — Sets whether the user can open multiple files (and directories) at a time.

– (void)**setCanChooseDirectories:**(BOOL)*flag* — Sets whether the user can choose directories.

– (void)**setCanChooseFiles:**(BOOL)*flag* — Sets whether the user can choose files.

Querying the Chosen Files

– (NSArray *)**filenames** — Returns an array containing the names of the selected files and directories.

Running the NSOpenPanel

– (int)**runModalForTypes:**(NSArray *)*fileTypes* — Invokes the **runModalForDirectory:file:types:** method, using the last directory from which a file was chosen as the path argument. Returns the value returned by that method.

– (int)**runModalForDirectory:**(NSString *)*path*
file:(NSString *)*filename*
types:(NSArray *)*fileTypes* — Displays the panel and begins its event loop. The panel displays the files in *path* that match the types in *fileTypes* (an array of NSString objects), with *filename* selected. Returns NSOKButton (if the user clicks the OK button) or NSCancelButton (if the user clicks the Cancel button).

NSPageLayout

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSPageLayout.h

Class Description

NSPageLayout is a type of NSPanel that queries the user for information such as paper type and orientation. This information is stored in an NSPrintInfo object, and is later used when printing. The NSPageLayout panel is created, displayed, and run (in a modal loop) when a **runPageLayout:** message is sent to the NSApplication object. By default, this message is sent up the responder chain when the user clicks the Page Layout menu item.

Typically, you access an NSPageLayout panel by invoking the **pageLayout** method. When the class receives a **pageLayout** message, it tries to reuse an existing panel rather than create a new one. If a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because Page Layout panels may be reused, you shouldn't modify the instance returned by **pageLayout**, except through the methods listed below. If you must modify the Page Layout panel in other ways than those allowed by its methods, create and manage your own instance using the **alloc...** and **init...** methods rather than the **pageLayout** method.

You can add your own controls to the Page Layout panel through the **setAccessoryView:** method. The panel is automatically resized to accommodate the NSView that you've added. Note that you can't retrieve the NSPageLayout's settings through messages to the page layout panel object—NSPageLayout does not have accessor methods to obtain the state of its controls. If controls you add through an accessory view need to know the values of the existing controls in the page layout panel (or vice versa), access NSPageLayout's controls using the tags defined in AppKit/NSPageLayout.h as arguments to **viewWithTag:** messages to the page layout panel object. Controls thus returned can then be queried for their state.

Creating an NSPageLayout Instance

+ (NSPageLayout *)**pageLayout** — Returns a default NSPageLayout object.

Running the Panel

– (int)**runModal** — Displays the panel and begins its event loop. The panel's values are recorded in the shared NSPrintInfo object.

– (int)**runModalWithPrintInfo:**(NSPrintInfo *)*pInfo* — Displays the panel and begins its event loop. The panel's values are recorded in the *pInfo*, the supplied NSPrintInfo object.

Customizing the Panel

– (NSView *)**accessoryView**	Returns the NSPageLayout's accessory View.
– (void)**setAccessoryView:**(NSView *)*aView*	Adds a View to the panel.

Updating the Panel's Display

– (void)**convertOldFactor:**(float *)*old* **newFactor:**(float *)*new*	Returns by reference the ratio between a point and the currently chosen unit of measurement. If invoked within the **pickedUnits:** method, *old* refers to the ratio before the user's choice and *new* refers to the new ratio.
– (void)**pickedButton:**(id)*sender*	Stops the event loop.
– (void)**pickedOrientation:**(id)*sender*	Updates the panel with the selected orientation.
– (void)**pickedPaperSize:**(id)*sender*	Updates the panel when a paper size is selected.
– (void)**pickedUnits:**(id)*sender*	Updates the panel when a new unit is selected.

Communicating with the NSPrintInfo Object

– (NSPrintInfo *)**printInfo**	Returns the NSPrintInfo object that used when the panel is run.
– (void)**readPrintInfo**	Reads the NSPageLayout's values from the NSPrintInfo object.
– (void)**writePrintInfo**	Writes the NSPageLayout's values to the NSPrintInfo object.

NSPanel

Inherits From: NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSPanel.h

Class Description

The NSPanel class defines objects that manage the panels of the OpenStep user interface. A panel is a window that serves an auxiliary function within an application. It generally displays controls that the user can act on to give instructions to the application or to modify the contents of a standard window.

Panels behave differently from standard windows in only a small number of ways, but the ways are important to the user interface:

- Panels can assume key window—but not main window—status. (The key window receives keyboard events. The main window is the primary focus of user actions; it might contain the document the user is working on, for example.)
- On-screen panels are normally removed from the screen list when the user begins to work in another application, and are restored to the screen when the user returns to the panel's application.

To aid in their auxiliary role, panels can be assigned special behaviors:

- A panel can be precluded from becoming the key window until the user makes a selection (makes some view in the panel the first responder) indicating an intention to begin typing. This prevents key window status from shifting to the panel unnecessarily.
- Palettes and similar panels can be made to float above standard windows and other panels. This prevents them from being covered and keeps them readily available to the user.
- A panel can be made to work—to receive mouse and keyboard events—even when there's an attention panel on-screen. This permits actions within the panel to affect the attention panel.

Determining the Panel Behavior

– (BOOL)**becomesKeyOnlyIfNeeded**	Returns whether the receiver waits to become key window.
– (BOOL)**isFloatingPanel**	Returns whether the receiver floats above other windows.
– (void)**setBecomesKeyOnlyIfNeeded:**(BOOL)*flag*	Sets whether the receiver waits to become key window.
– (void)**setFloatingPanel:**(BOOL)*flag*	Sets whether the receiver floats above other windows.
– (void)**setWorksWhenModal:**(BOOL)*flag*	Sets whether the receiver can operate even when an attention panel is on-screen.
– (BOOL)**worksWhenModal**	Returns whether the receiver can operate even when an attention panel is on-screen. The default is NO.

NSPasteboard

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSPasteboard.h

Class Description

NSPasteboard objects transfer data to and from the pasteboard server. The server is shared by all running applications. It contains data that the user has cut or copied and may paste, as well as other data that one application wants to transfer to another. NSPasteboard objects are an application's sole interface to the server and to all pasteboard operations.

Named Pasteboards

Data in the pasteboard server is associated with a name that indicates how it's to be used. Each set of data and its associated name is, in effect, a separate pasteboard, distinct from the others. An application keeps a separate NSPasteboard object for each named pasteboard that it uses. There are five standard pasteboards in common use:

General pasteboard	The pasteboard that's used for ordinary cut, copy, and paste operations. It holds the contents of the last selection that's been cut or copied.
Font pasteboard	The pasteboard that holds font and character information and supports the Copy Font and Paste Font commands.
Ruler pasteboard	The pasteboard that holds information about paragraph formats in support of the Copy Ruler and Paste Ruler commands.
Find pasteboard	The pasteboard that holds information about the current state of the active application's Find panel. This information permits users to enter a search string into the Find panel, then switch to another application to conduct the search.
Drag pasteboard	The pasteboard that stores data to be manipulated as the result of a drag operation.

Each standard pasteboard is identified by a unique name (stored in global string objects):

NSGeneralPboard
NSFontPboard
NSRulerPboard
NSFindPboard
NSDragPboard

You can create private pasteboards by asking for an NSPasteboard object with any name other than those listed above. The name of a private pasteboard can be passed to other applications to allow them to share the data it holds.

The NSPasteboard class makes sure there's never more than one object for each named pasteboard. If you ask for a new object when one has already been created for the pasteboard with that name, the existing object will be returned to you.

Data Types

Data can be placed in the pasteboard server in more than one representation. For example, an image might be provided both in Tag Image File Format (TIFF) and as encapsulated PostScript code (EPS). Multiple representations give pasting applications the option of choosing which data type to use. In general, an application taking data from the pasteboard should choose the richest representation it can handle—rich text over plain ASCII, for example. An application putting data in the pasteboard should promise to supply it in as many data types as possible, so that as many applications as possible can make use of it.

Data types are identified by string objects containing the full type name. These global variables identify the string objects for the standard pasteboard types:

Type	Description
NSStringPboardType	NSString data
NSPostScriptPboardType	Encapsulated PostScript code (EPS)
NSTIFFPboardType	Tag Image File Format (TIFF)
NSRTFPboardType	Rich Text Format (RTF)
NSFilenamesPboardType	ASCII text designating one or more file names
NSTabularTextPboardType	Tab-separated fields of ASCII text
NSFontPboardType	Font and character information
NSRulerPboardType	Paragraph formatting information
NSFileContentsPboardType	A representation of a file's contents
NSColorPboardType	NSColor data
NSGeneralPboardType	Describes a selection
NSDataLinkPboardType	Defines a link between documents

Types other than those listed can also be used. For example, your application may keep data in a private format that's richer than any of the types listed above. That format can also be used as a pasteboard type.

Reading and Writing Data

Typically, data is written to the pasteboard using **setData:forType:** and read using **dataForType:**. However, data of the type NSFileContentsPboardType, representing the contents of a named file, must be written to the NSPasteboard object using **writeFileContents:** and copied from the object to a file using **readFileContentsType:toFile:**.

Errors

Except where errors are specifically mentioned in the method descriptions, any communications error with the pasteboard server raises an NSPasteboardCommunicationException exception.

Creating and Releasing an NSPasteboard Object

+ (NSPasteboard *)**generalPasteboard** Returns the general NSPasteboard.

+ (NSPasteboard *)**pasteboardWithName:**(NSString *)*name*
Returns the NSPasteboard named *name*.

+ (NSPasteboard *)**pasteboardWithUniqueName** Returns a uniquely named NSPasteboard.

– (void)**releaseGlobally** Releases the NSPasteboard and its resources in the pasteboard server.

Getting Data in Different Formats

+ (NSPasteboard *)**pasteboardByFilteringData:**(NSData *)*data*
ofType:(NSString *)*type* Returns an NSPasteboard that contains data of all types filterable from *data* of type *type*.

+ (NSPasteboard *)**pasteboardByFilteringFile:**(NSString *)*filename*
Returns an NSPasteboard that contains data of all types filterable from *filename*.

+ (NSPasteboard *)**pasteboardByFilteringTypesInPasteboard:**(NSPasteboard *)*pboard*
Returns an NSPasteboard that contains data of all types filterable from *pboard*.

+ (NSArray *)**typesFilterableTo:**(NSString *)*type* Returns an array specifying all types *type* can be filtered to.

Referring to a Pasteboard by Name

– (NSString *)**name** Returns the NSPasteboard's name.

Writing Data

– (int)**addTypes:**(NSArray *)*newTypes*
owner:(id)*newOwner* Adds data types to the NSPasteboard and declares a new owner. Returns the new change count or 0 in case of error.

– (int)**declareTypes:**(NSArray *)*newTypes*
owner:(id)*newOwner* Sets the data types and owner of the NSPasteboard and returns the new change count.

– (BOOL)**setData:**(NSData *)*data*
forType:(NSString *)*dataType* Writes data of type *dataType* to the pasteboard server from *data*. Returns YES if the data is successfully written; otherwise returns NO.

– (BOOL)**setPropertyList:**(id)*propertyList*
forType:(NSString *)*dataType* Writes data of type *dataType* to the pasteboard server from *propertyList*. Returns YES if the data is successfully written; otherwise returns NO.

– (BOOL)**setString:**(NSString *)*string*
    forType:(NSString *)*dataType*

Writes data of type *dataType* to the pasteboard server from *string*. Returns YES if the data is successfully written; otherwise returns NO.

– (BOOL)**writeFileContents:**(NSString *)*filename*

Writes data from *filename* to the pasteboard server.

Determining Types

– (NSString *)**availableTypeFromArray:**(NSArray *)*types*

Returns first type in *types* that matches a type declared in the receiver.

– (NSArray *)**types**

Returns an array of the NSPasteboard's data types.

Reading Data

– (int)**changeCount**

Returns the NSPasteboard's change count.

– (NSData *)**dataForType:**(NSString *)*dataType*

Returns NSPasteboard data using the type specified by *dataType*.

– (id)**propertyListForType:**(NSString *)*dataType*

Returns a property list object using the type specified by *dataType*.

– (NSString *)**readFileContentsType:**(NSString *)*type*
    toFile:(NSString *)*filename*

Reads data of type *type* representing a file's contents from the NSPasteboard and writes it to *filename*. Returns the actual name of the file that was written.

– (NSString *)**stringForType:**(NSString *)*dataType*

Returns an NSString using the type specified by *dataType*.

Methods Implemented by the Owner

– (void)**pasteboard:**(NSPasteboard *)*sender*
    provideDataForType:(NSString *)*type*

Implemented to write promised data to *sender* as *type*.

– (void)**pasteboardChangedOwner:**(NSPasteboard *)*sender*

Notifies prior owner that ownership changed.

NSPopUpButton

Inherits From: NSButton : NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSPopUpButton.h

Class Description

The NSPopUpButton class defines objects that implement the pop-up and pull-down lists of the OpenStep graphical user interface. When configured to display a pop-up list, an NSPopUpButton contains a number of options and displays as its title the option that was last selected. A pop-up list is often used for selecting items from a small- to medium-sized set of options (like the zoom factor for a document window). It's a useful alternative to a matrix of radio buttons or an NSBrowser when screen space is at a premium; a zoom factor pop-up can easily fit next to a scroll bar at the bottom of a window, for example.

When configured to display a pull-down list, an NSPopUpButton is generally used for selecting commands in a very specific context. You can think of a pull-down list as a compact form of menu. A pull-down list's title isn't affected by the user's actions, and a pull-down list always displays a title that identifies the type of commands it contains. When the commands only make sense in the context of a particular display, a pull-down list can be used in that display to keep the related actions nearby, and to keep them out of the way when that display isn't visible.

Initializing an NSPopUpButton

– (id)**initWithFrame:**(NSRect)*frameRect*
pullsDown:(BOOL)*flag*

Initializes a newly allocated NSPopUpButton, giving it the frame specified by *frameRect.* If *flag* is YES, the receiver is initialized to operate as a pull-down list; otherwise, it operates as a pop-up list.

Target and Action

– (SEL)**action**

Returns the NSPopUpButton's action method.

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

Sets the NSPopUpButton's action method to *aSelector.*

Adding Items

– (void)**addItemWithTitle:**(NSString *)*title*

Adds an item with *title* as its title to the end of the item list.

– (void)**addItemsWithTitles:**(NSArray *)*itemTitles*

Adds multiple items to the end of the item list. The titles for the new items are taken from the *itemTitles* array.

– (void)**insertItemWithTitle:**(NSString *)*title* **atIndex:**(unsigned int)*index* — Inserts an item with *title* as its title at position *index*.

Removing Items

– (void)**removeAllItems** — Removes all items in the receiver's item list.

– (void)**removeItemWithTitle:**(NSString *)*title* — Removes the item whose title matches *title.*

– (void)**removeItemAtIndex:**(int)*index* — Removes the item at the specified index.

Querying the NSPopUpButton about Its Items

– (int)**indexOfItemWithTitle:**(NSString *)*title* — Returns the index of the item whose title matches *title*, or –1 if no match is found.

– (int)**indexOfSelectedItem** — Returns the index of the item last selected by the user, or –1 if there's no selected item.

– (int)**numberOfItems** — Returns the number of items in the receiver's item list.

– (NSMenuCell *)**itemAtIndex:**(int)*index* — Returns the NSMenuCell for the item at *index*, or **nil** if no such item exists.

– (NSMatrix *)**itemMatrix** — Returns the NSMatrix that holds the receiver's items.

– (NSString *)**itemTitleAtIndex:**(int)*index* — Returns the title of the item at *index*, or the empty string if no such item exists.

– (NSArray *)**itemTitles** — Returns an NSArray that holds the titles of the receiver's items.

– (NSMenuCell *)**itemWithTitle:**(NSString *)*title* — Returns the NSMenuCell for the item whose title is *title*, or **nil** if no such item exists

– (NSMenuCell *)**lastItem** — Returns the NSMenuCell corresponding to the last item in the list.

– (NSMenuCell *)**selectedItem** — Returns the NSMenuCell for the selected item.

– (NSString *)**titleOfSelectedItem** — Returns the title of the item last selected by the user, or the empty string if there's no such item.

Manipulating the NSPopUpButton

– (NSFont *)**font** — Returns the font used to draw the items.

– (BOOL)**pullsDown** — Returns YES if the receiver is configured as a pull-down list, and NO if it's configured as a pop-up list.

– (void)**selectItemAtIndex:**(int)*index* — Selects the item at *index* and invokes **synchronizeTitleAndSelectedItem**.

– (void)**selectItemWithTitle:**(NSString *)*title* — Selects the item whose title is *title* and invokes **synchronizeTitleAndSelectedItem**.

– (void)**setFont:**(NSFont *)*fontObject* — Sets the font used to draw the items.

– (void)**setPullsDown:**(BOOL)*flag* — If *flag* is YES, the receiver is configured as a pull-down list. If *flag* is NO, the receiver is configured as a pop-up list.

– (void)**setTarget:**(id)*anObject* — Sets the target for action messages to *anObject*.

– (void)**setTitle:**(NSString *)*aString* — Adds a new item (if the receiver doesn't already have an item titled *aString*), makes it the selected item, and invokes **synchronizeTitleAndSelectedItem**.

– (NSString *)**stringValue** — Returns the title of the selected item.

– (void)**synchronizeTitleAndSelectedItem** — Ensures that the receiver's title agrees with the title of the selected item (see **indexOfSelectedItem**). If there's no selected item, this method selects the first item in the item list and sets the receiver's title to match. This method is useful in subclasses that directly select items in the item matrix or that override **setTitle:**.

– (id)**target** — Returns the target for action messages.

Displaying the NSPopUpButton's Items

– (BOOL)**autoenablesItems** — Returns whether the NSPopUpButton enables and disables its items. (See the NSMenuActionResponder informal protocol.)

– (void)**setAutoenablesItems:**(BOOL)*flag* — Sets whether the NSPopUpButton enables and disables its items. (See the NSMenuActionResponder informal protocol.)

NSPrinter

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSPrinter.h

Class Description

An NSPrinter object describes a printer's capabilities, such as whether the printer can print in color and whether it provides a particular font. An NSPrinter object represents either a particular make or type of printer, or an actual printer available to the computer.

There are two ways to create an NSPrinter:

- To create an abstract object that provides information about a type of printer rather than an object that represents an actual printer device, use the **printerWithType:** class method, passing a printer type (an NSString) as the argument. The **printerTypes** class method provides a list of the printer types recognized by the computer. Printer types are described in files written in PostScript Printer Description (PPD) format. The location of these files is platform dependent.

- To create or find an NSPrinter that corresponds to an actual printer device, use the **printerWithName:** class method, passing the name of a printer. The way you find out what the available printer names are depends on the platforms you are using.

Once you have an NSPrinter, there's only one thing you can do with it: Retrieve information regarding the type of printer or regarding the actual printer the object represents. You can't change the information in an NSPrinter, nor can you use an NSPrinter to initiate or control a printing job.

When you create an NSPrinter object, the object reads the file that corresponds to the type of printer you specified and stores the data it finds there in named tables. Printer types are described in files written in the PostScript Printer Description (PPD) format. Any piece of information in the PPD tables can be retrieved through the methods **stringForKey:inTable:** and **stringListForKey:inTable:**, as explained later. Commonly needed items, such as whether a printer is color or the size of the page on which it prints, are available through more direct methods (methods such as **isColor** and **pageSizeForPaper:**).

Note: To understand what the NSPrinter tables contain, you need to be acquainted with the PPD file format. This is described in *PostScript Printer Description File Format Specification, version 4.0,* available from Adobe Systems Incorporated. The rest of this class description assumes a familiarity with the concepts and terminology presented in the Adobe manual. A brief summary of the PPD format is given below; PPD terms defined in the Adobe manual are shown in italic.

PPD Format

A PPD file statement, or *entry*, associates a *value* with a *main keyword*:

> **mainKeyword*: *value*

The asterisk is literal; it indicates the beginning of a new entry.

For example:

```
*ModelName: "MMimeo Machine"
*3dDevice: False
```

A main keyword can be qualified by an *option keyword*:

> **mainKeyword optionKeyword*: *value*

For example:

```
*PaperDensity Letter: "0.1"
*PaperDensity Legal: "0.2"
*PaperDensity A4: "0.3"
*PaperDensity B5: "0.4"
```

In addition, any number of entries may have the same main keyword with no option keyword yet give different values:

```
*InkName: ProcessBlack/Process Black
*InkName: CustomColor/Custom Color
*InkName: ProcessCyan/Process Cyan
*InkName: ProcessMagenta/Process Magenta
*InkName: ProcessYellow/Process Yellow
```

Option keywords and values can sport *translation strings*. A translation string is a textual description, appropriate for display in a user interface, of the option or value. An option or value is separated from its translation string by a slash:

```
*Resolution 300dpi/300 dpi: " ... "
*InkName: ProcessBlack/Process Black
```

In the first example, the **300dpi** option would be presented in a user interface as "300 dpi." The second example assigns the string "Process Black" as the translation string for the **ProcessBlack** value.

NSPrinter treats entries that have an ***OrderDependency** or ***UIConstraint** main keyword specially. Such entries take the following forms (the bracketed elements are optional):

> *OrderDependency: *real section mainKeyword* [*optionKeyword*]
> *UIConstraint: *mainKeyword1* [*optionKeyword1*] *mainKeyword2* [*optionKeyword2*]

There may be more than one UIConstraint entry with the same *mainKeyword1* or *mainKeyword1/optionKeyword1* value. Below are some examples of ***OrderDependency** and ***UIConstraint** entries:

```
*OrderDependency: 10 AnySetup *Resolution
*UIConstraint: *Option3 None *PageSize Legal
*UIConstraint: *Option3 None *PageRegion Legal
```

Explaining these entries is beyond the scope of this documentation; however, it's important to note their forms in order to understand how they're represented in the NSPrinter tables.

NSPrinter Tables

NSPrinter defines five key-value tables to store PPD information. The tables are identified by the names given below:

Name	Contents
PPD	General information about a printer type. This table contains the values for all entries in a PPD file except those with the ***OrderDependency** and ***UIConstraint** main keywords. The values in this table don't include the translation strings.
PPDOptionTranslation	Option keyword translation strings.
PPDArgumentTranslation	Value translation strings.
PPDOrderDependency	***OrderDependency** values.
PPDUIConstraints	***UIConstraint** values.

There are two principle methods for retrieving data from the NSPrinter tables:

- **stringForKey:inTable:** returns the value for the first occurrence of a given key in the given table.
- **stringListForKey:inTable:** returns an array of values, one for each occurrence of the key.

For both methods, the first argument is an NSString that names a key—which part of a PPD file entry the key corresponds to depends on the table (as explained in the following sections). The second argument names the table that you want to look in. The values that are returned by these methods, whether singular or in an array, are always NSStrings, even if the value wasn't a quoted string in the PPD file.

The NSPrinter tables store data as ASCII text, thus the two methods described above are sufficient for retrieving any value from any table. NSPrinter provides a number of other methods, such as **booleanForKey:inTable:** and **intForKey:inTable:**, that retrieve single values and coerce them, if possible, into particular data types. The coercion doesn't affect the data that's stored in the table (it remains in ASCII format).

To check the integrity of a table, use the **isKey:forTable:** and **statusForTable:** methods. The former returns a boolean that indicates whether the given key is valid for the given table; the latter returns an error code that describes the general state of a table (in particular, whether it actually exists).

Retrieving Values from the PPD Table

Keys for the PPD table are strings that name a main keyword or main keyword/option keyword pairing (formatted as “*mainKeyword*/*optionKeyword*”). In both cases, you exclude the main keyword asterisk. The following example creates an NSPrinter and invokes **stringForKey:inTable:** to retrieve the value for an un-optioned main keyword:

```
/* Create an NSPrinter object for a printer type. */
NSPrinter *prType = [NSPrinter
                     printerWithType:@"My_Mimeo_Machine"]

 NSString *sValue = [prType stringForKey:@"3dDevice" inTable:@"PPD"];
/* sValue is "False". */
```

To retrieve the value for a main keyword/option keyword pair, pass the keywords formatted as “*mainKeyword*/*optionKeyword*”:

```
NSString *sValue = [prType stringForKey:@"PaperDensity/A4"
                          inTable:@"PPD"];
/* sValue is "0.3". */
```

stringForKey:inTable: can determine if a main keyword has options. If you pass a main keyword (only) as the first argument to the method, and if that keyword has options in the PPD file, the method returns the empty string. If it doesn’t have options, it returns the value of the first occurrence of the main keyword:

```
NSString *sValue = [prType stringForKey:@"PaperDensity" inTable:@"PPD"];
/* sValue is empty string*/

NSString *sValue = [prType stringForKey:@"InkName" inTable:@"PPD"];
/* sValue is "ProcessBlack" */
```

To retrieve the values for all occurrences of an un-optioned main keyword, use the **stringListForKey:inTable:** method:

```
NSArray *sList = [prType stringListForKey:@"InkName" inTable:@"PPD"];
/* [slist objectAtIndex:0] is "ProcessBlack",
   [slist objectAtIndex:1] is "CustomColor",
   [slist objectAtIndex:2] is "ProcessCyan", and so on. */
```

In addition, **stringListForKey:inTable:** can be used to retrieve all the options for a main keyword (given that the main keyword has options):

```
NSArray *sList = [prType stringListForKey:@"PaperDensity"
                          inTable:@"PPD"];
/* [slist objectAtIndex:0] is "Letter",
   [slist objectAtIndex:1] is "Legal",
   [slist objectAtIndex:2] is "A4", and so on. */
```

Retrieving Values from the Option and Argument Translation Tables

A key to a translation table is like that to the PPD table: It's a main keyword or main/option keyword pair (again excluding the asterisk). However, the values that are returned from the translation tables are the translation strings for the option or argument (value) portions of the PPD file entry. For example:

```
NSString *sValue = [prType stringForKey:@"Resolution/300dpi"
                          inTable:@"PPDOptionTranslation"];
/* sValue is "300 dpi". */

NSArray *sList = [prType stringListForKey:@"InkName"
                          inTable:@"PPDArgumentTranslation"];
/* [slist objectAtIndex:0] is "Process Black",
   [slist objectAtIndex:1] is "Custom Color",
   [slist objectAtIndex:2] is "Process Cyan", and so on. */
```

As with the PPD table, requesting an NSArray of NSStrings for an un-optioned main keyword returns the keyword's options (if it has any).

Retrieving Values from the Order Dependency Table

As mentioned earlier, an order dependency entry takes this form:

*OrderDependency: *real section mainKeyword* [*optionKeyword*]

These entries are stored in the PPDOrderDependency table. To retrieve a value from this table, always use **stringListForKey:inTable:**. The value passed as the key is, again, a main keyword or main keyword/option keyword pair; however, these values correspond to the *mainKeyword* and *optionKeyword* parts of an order dependency entry's value. As with the other tables, the main keyword's asterisk is excluded. The method returns an NSArray of two NSStrings that correspond to the *real* and *section* values for the entry. For example:

```
NSArray *sList = [prType stringListForKey:@"Resolution"
                          inTable:@"PPDOrderDependency"]
/* [slist objectAtIndex:0] = "10", [slist objectAtIndex:1] = "AnySetup" */
```

Retrieving Values from the UIConstraints Table

Retrieving a value from the PPDUIConstraints table is similar to retrieving a value from the PPDOrderDependency table: always use **stringListForKey:inTable:** and the key corresponds to elements in the entry's value. Given the following form (as described earlier), the key corresponds to *mainKeyword1*/*optionKeyword1*:

*UIConstraint: *mainKeyword1* [*optionKeyword1*] *mainKeyword2* [*optionKeyword2*]

The NSArray that's returned by **stringListForKey:inTable:** contains the *mainKeyword2* and *optionKeyword2* values (with the keywords stored as separate elements in the NSArray) for every ***UIConstraints** entry that has the given *mainKeyword1*/*optionKeyword1* value. For example:

```
NSArray *sList = [prType stringListForKey:@"Option3/None"
                          inTable:@"PPDUIConstraints"]
/* [slist objectAtIndex:0] = "PageSize", [slist objectAtIndex:1] = "Legal",
   [slist objectAtIndex:2] = "PageRegion", [slist objectAtIndex:3] = "Legal" */
```

Note that the main keywords that are returned in the NSArray don't have asterisks. Also, the NSArray that's returned always alternates main and option keywords. If a particular main keyword doesn't have an option associated with it, the string for the option will be empty (but the entry in the NSArray for the option *will* exist).

Finding an NSPrinter

+ (NSPrinter *)**printerWithName:**(NSString *)*name* Returns the NSPrinter with the given name.

+ (NSPrinter *)**printerWithType:**(NSString *)*type* Returns an NSPrinter object for the given printer type.

+ (NSArray *)**printerTypes** Returns the recognized printer types.

Printer Attributes

– (NSString *)**host** Returns the name of the printer's host computer.

– (NSString *)**name** Returns the printer's name.

– (NSString *)**note** Returns the note associated with the printer.

– (NSString *)**type** Returns the name of the printer's type.

Retrieving Specific Information

– (BOOL)**acceptsBinary** Returns YES if the printer accepts binary PostScript.

– (NSRect)**imageRectForPaper:**(NSString *)*paperName*
Returns the printing rectangle for the named paper type. Possible values for *paperName* are contained in the printer's PPD file. Typical values are Letter and Legal.

– (NSSize)**pageSizeForPaper:**(NSString *)*paperName*
Returns the size of the page for the named paper type.

– (BOOL)**isColor** Returns whether the printer can print color.

– (BOOL)**isFontAvailable:**(NSString *)*fontName* Returns whether the named font is available to the printer.

– (int)**languageLevel** Returns the PostScript Language Level recognized by the printer.

– (BOOL)**isOutputStackInReverseOrder** Returns whether the printer outputs pages in reverse page order.

Querying the NSPrinter Tables

– (BOOL)**booleanForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns a boolean value associated with *key* in *table*.
– (NSDictionary *)**deviceDescription**	Returns a dictionary of keys and values describing the device. See NSGraphics.h for possible keys.
– (float)**floatForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns a floating-point value associated with *key* in *table*.
– (int)**intForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns an integer value associated with *key* in *table*.
– (NSRect)**rectForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns rectangle associated with *key* in *table*.
– (NSSize)**sizeForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns the size associated with *key* in *table*.
– (NSString *)**stringForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns a string associated with *key* in *table*.
– (NSArray *)**stringListForKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns an array of strings associated with *key* in *table*.
– (NSPrinterTableStatus)**statusForTable:**(NSString *)*table*	Returns the status (NSPrinterTableOK, NSPrinterTableNotFound, NSPrinterTableError) of the given table.
– (BOOL)**isKey:**(NSString *)*key* **inTable:**(NSString *)*table*	Returns whether *key* is a key in *table*.

NSPrintInfo

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSPrintInfo.h

Class Description

An NSPrintInfo object stores information that's used during printing. A shared NSPrintInfo object is automatically created for an application and is used by default for all printing jobs for that application. You can create any number of additional NSPrintInfo objects; however, only one can be "active" at a time, as set through the **setSharedPrintInfo:** class method. The shared NSPrintInfo object is returned through the **sharedPrintInfo** class method.

An NSPrintInfo object is used by the NSPrintOperations class to control printing. If you create special instances of NSPrintInfo objects for a specific printing task, you must ensure that either the application's shared NSPrintInfo object is current, or you must instantiate an NSPrintOperations object using one of its methods that explicitly designate an NSPrintInfo object.

Although you can set an NSPrintInfo's attributes through the methods it provides, this is usually the task of other objects, notably the NSPageLayout and NSPrintPanel objects. The NSView or NSWindow that's being printed may also supercede some NSPrintInfo settings. In particular, a NSView or NSWindow can supply the range of pages in the document and can provide its own pagination mechanism through the **knowsPagesFirst:last:** and **rect:forPage:** methods (see the documentation of these methods in the NSView class for details).

If the printed NSView or NSWindow doesn't supply a pagination, the NSPrintInfo's vertical and horizontal pagination constants are used to trigger built-in pagination mechanisms:

Pagination Constant	Meaning
NSAutoPagination	The image is diced into equal-sized rectangles and placed in one column of pages.
NSFitPagination	The image is scaled to produce one column or one row of pages.
NSClipPagination	The image is clipped to produce one column or row of pages.

Vertical and horizontal pagination needn't be the same. However, if either dimension is scaled (NSFitPagination), the other dimension is scaled by the same amount to avoid stretching the image. If both dimensions are scaled, the scaling factor that produces the smallest image is used. Note that NSPrintInfo's scaling factor is independent of the scaling that's imposed by pagination and is applied after the document has been paginated.

NSPrintInfo uses points as the unit of measurement for paper size and margin width in the methods below. See the NSFont specification for a discussion of points.

Creating and Initializing an NSPrintInfo Instance

– (id)**initWithDictionary:**(NSDictionary *)*aDict*	Initializes a newly allocated NSPrintInfo object by assigning it the parameters specified in *aDict*. This is the designated initializer for the class.

Managing the Shared NSPrintInfo Object

+ (void)**setSharedPrintInfo:**(NSPrintInfo *)*printInfo*	Sets the shared NSPrintInfo object to *printInfo*.
+ (NSPrintInfo *)**sharedPrintInfo**	Returns the shared NSPrintInfo object.

Managing the Printing Rectangle

+ (NSSize)**sizeForPaperName:**(NSString *)*name*	Returns the size for the specified type of paper. *name* identifies the type of paper, such as “Letter” or “Legal”. Paper names are implementation specific.
– (float)**bottomMargin**	Returns the height of the bottom margin.
– (float)**leftMargin**	Returns the width of the left margin.
– (NSPrintingOrientation)**orientation**	Returns whether the orientation is Portrait or Landscape.
– (NSString *)**paperName**	Returns the paper type, such as “Letter” or “Legal”. Paper names are implementation specific.
– (NSSize)**paperSize**	Returns the size of the paper.
– (float)**rightMargin**	Returns the width of the right margin.
– (void)**setBottomMargin:**(float)*value*	Sets the bottom margin to *value*.
– (void)**setLeftMargin:**(float)*value*	Sets the left margin to *value*.
– (void)**setOrientation:**(NSPrintingOrientation)*mode*	Sets the orientation as Portrait or Landscape.
– (void)**setPaperName:**(NSString *)*name*	Sets the paper type. *name* identifies the type of paper, such as “Letter” or “Legal”. Paper names are implementation specific.
– (void)**setPaperSize:**(NSSize)*size*	Sets the width and height of the paper.
– (void)**setRightMargin:**(float)*value*	Sets the right margin to *value*.
– (void)**setTopMargin:**(float)*value*	Sets the top margin to *value*.
– (float)**topMargin**	Returns the height of the top margin.

Pagination

– (NSPrintingPaginationMode)**horizontalPagination** Returns the horizontal pagination mode.

– (void)**setHorizontalPagination:**(NSPrintingPaginationMode)*mode*
Sets the horizontal pagination mode.

– (void)**setVerticalPagination:**(NSPrintingPaginationMode)*mode*
Sets the vertical pagination mode.

– (NSPrintingPaginationMode)**verticalPagination** Returns the vertical pagination mode.

Positioning the Image on the Page

– (BOOL)**isHorizontallyCentered** Returns whether the image is centered horizontally.

– (BOOL)**isVerticallyCentered** Returns whether the image is centered vertically.

– (void)**setHorizontallyCentered:**(BOOL)*flag* Sets whether the image is centered horizontally.

– (void)**setVerticallyCentered:**(BOOL)*flag* Sets whether the image is centered vertically.

Specifying the Printer

+ (NSPrinter *)**defaultPrinter** Returns the user’s default printer.

+ (void)**setDefaultPrinter:**(NSPrinter *)*printer* Sets the user’s default printer.

– (NSPrinter *)**printer** Returns the NSPrinter that’s used for printing.

– (void)**setPrinter:**(NSPrinter *)*aPrinter* Sets the printer that’s used in subsequent printing jobs.

Controlling Printing

– (NSString *)**jobDisposition** Returns the action specified for the job: printing, faxing, previewing, etc. See **setJobDisposition:**.

– (void)**setJobDisposition:**(NSString *)*disposition* Sets the action specified for the job. *disposition* can be one of NSPrintSpoolJob, NSPrintFaxJob, NSPrintPreviewJob, NSPrintSaveJob, NSPrintCancelJob.

– (void)**setUpPrintOperationDefaultValues** Allows the receiver to set any attribute that hasn’t been previously set.

Accessing the NSPrintInfo Object’s Dictionary

– (NSMutableDictionary *)**dictionary** Returns the NSPrintInfo object’s dictionary.

NSPrintOperation

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSPrintOperation.h

Class Description

NSPrintOperation controls operations that generate Encapsulated PostScript (EPS) code or PostScript print jobs. Generally, EPS code is used to transfer images between applications, which happens when the user copies and pastes graphics, uses a Service, or uses ObjectLinks. PostScript print jobs are generated when the user prints and faxes documents. An NSPrintOperation does not generate PostScript code itself; it just controls the overall process, relying on an NSView object to generate the actual code.

NSPrintOperation relies mainly on two other objects: an NSPrintInfo object, which specifies how the code should be generated, and an NSView object, which performs the actual code generation. You specify these two objects in the method you use to create the NSPrintOperation. If no NSPrintInfo is specified, NSPrintOperation uses the shared NSPrintInfo, which contains default values. The shared NSPrintInfo works well for applications that are not document-based. Document-based applications should create an NSPrintInfo for each document that might be printed or copied and use that object instead.

You should create an NSPrintOperation in any method that is invoked when a user executes a Print command or a Copy command. That method also must send NSPrintOperation a **runOperation** message to start the operation. A **print:** method for a document-based application might look like this:

```
- (void)print:sender {
[[NSPrintOperation printOperationWithView:[self myView] printInfo:[document
docPrintInfo]] runOperation];
}
```

This method creates an NSPrintOperation for a print job that uses the document's NSPrintInfo. Because this is a print job, a Print panel (NSPrintPanel object) is displayed to allow the user to select printing options. The NSPrintOperation copies the NSPrintInfo, updates this copy with information from the Print panel, and uses the specified NSView to perform the operation.

The information stored in an NSPrintInfo that's retained between operations is information that's likely to remain constant for a document, such as its page size. All information that's likely to change between operations is set to a default value in the NSPrintInfo before the operation begins. In this way, even though NSPrintOperation updates the NSPrintInfo with information from the Print panel for print jobs, that information is reset back to the default values for each print job. Because NSPrintOperation keeps a copy of the NSPrintInfo it uses, you could duplicate a specific print job by storing that copy and reusing it.

Creating and Initializing an NSPrintOperation Object

+ (NSPrintOperation *)**EPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toData:(NSMutableData *)*data*

Returns a new NSPrintOperation that controls the copying of EPS graphics from the area specified by *rect* in *aView*, using the parameters in the default NSPrintInfo. The code is written to *data*. Raises NSPrintOperationExistsException if there is already a print operation in progress.

+ (NSPrintOperation *)**EPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toData:(NSMutableData *)*data*
printInfo:(NSPrintInfo *)*aPrintInfo*

Returns a new NSPrintOperation that controls the copying of EPS graphics from the area specified by *rect* in *aView*, using the parameters in *aPrintInfo*. The code is written to *data*. Raises NSPrintOperationExistsException if there is already a print operation in progress.

+ (NSPrintOperation *)**EPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toPath:(NSString *)*path*
printInfo:(NSPrintInfo *)*aPrintInfo*

Returns a new NSPrintOperation that controls the copying of EPS graphics from the area specified by *rect* in *aView*, using the parameters in *aPrintInfo*. The code is written to *path*. Raises NSPrintOperationExistsException if there is already a print operation in progress.

+ (NSPrintOperation *)**printOperationWithView:**(NSView *)*aView*

Returns a new NSPrintOperation that controls the printing of *aView*, using the parameters in the shared NSPrintInfo object. Raises NSPrintOperationExistsException if there is already a print operation in progress.

+ (NSPrintOperation *)**printOperationWithView:**(NSView *)*aView*
printInfo:(NSPrintInfo *)*aPrintInfo*

Returns a new NSPrintOperation that controls the printing of *aView*, using the parameters in *aPrintInfo*. Raises NSPrintOperationExistsException if there is already a print operation in progress.

– (id)**initEPSOperationWithView:**(NSView *)*aView*
insideRect:(NSRect)*rect*
toData:(NSMutableData *)*data*
printInfo:(NSPrintInfo *)*aPrintInfo*

Initializes a newly allocated NSPrintOperation to control the copying of EPS graphics from the area specified by *rect* in *aView*, using the parameters in *aPrintInfo*. The code is written to *data*.

– (id)**initWithView:**(NSView *)*aView*
printInfo:(NSPrintInfo *)*aPrintInfo*

Initializes a newly allocated NSPrintOperation to control the printing of *aView*, using the parameters in *aPrintInfo*.

Setting the Print Operation

+ (NSPrintOperation *)**currentOperation** — Returns the NSPrintOperation that represents the current operation or **nil** if there is no such operation.

+ (void)**setCurrentOperation:**(NSPrintOperation *)*operation* — Sets the NSPrintOperation that represents the current operation.

Determining the Type of Operation

– (BOOL)**isEPSOperation** — Returns YES if the receiver controls an EPS operation and NO if the receiver controls a printing operation.

Controlling the User Interface

– (NSPrintPanel *)**printPanel** — Returns the NSPrintPanel object that's used when the operation is run.

– (BOOL)**showPanels** — Returns whether the Print panel will appear when the operation is run.

– (void)**setPrintPanel:**(NSPrintPanel *)*panel* — Sets the NSPrintPanel object that's used when the operation is run.

– (void)**setShowPanels:**(BOOL)*flag* — Sets whether the Print panel appears when the operation is run.

Managing the DPS Context

– (NSDPSContext *)**createContext** — Used by the NSPrintOperation object to create the DPS context for output generation, using the current NSPrintInfo settings.

– (NSDPSContext *)**context** — Returns the DPS context used for the receiver's operation.

– (void)**destroyContext** — Used by the NSPrintOperation object to destroy the DPS context at the end of the operation.

Page Information

– (int)**currentPage** — Returns the page number of the page being printed.

– (NSPrintingPageOrder)**pageOrder** — Returns the order in which pages will be printed.

– (void)**setPageOrder:**(NSPrintingPageOrder)*order* — Sets the order in which pages will be printed.

Running a Print Operation

– (void)**cleanUpOperation**	Invoked at end of an operation's run to set the current operation to **nil**.
– (BOOL)**deliverResult**	Delivers the results generated by **runOperation** to the intended destination: the print spooler, preview application, etc. Returns YES upon successful delivery and NO otherwise.
– (BOOL)**runOperation**	Causes the operation (copying EPS graphics or printing) to take place. Returns YES upon successful completion and NO otherwise.

Getting the NSPrintInfo Object

– (NSPrintInfo *)**printInfo**	Returns the receiver's NSPrintInfo object.
– (void)**setPrintInfo:**(NSPrintInfo *)*aPrintInfo*	Sets the receiver's NSPrintInfo object to *aPrintInfo*.

Getting the NSView Object

– (NSView *)**view**	Returns the NSView object that performs the operation controlled by the receiving object.

NSPrintPanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSPrintPanel.h

Class Description

NSPrintPanel creates a Print panel. The Print panel queries the user for information about a print job, such as which pages to print and how many copies.

When a **print:** message is sent to an NSView or NSWindow, an NSPrintOperation object is created to control the print operation, which includes deciding whether or not to use an NSPrintPanel. The NSPrintPanel will be used unless the **setShowPanels:NO** message is sent to the NSPrintOperation. If you're subclassing NSPrintPanel, send the **setPrintPanel** message to the NSPrintOperation object to ensure that an instance of your subclass is the unique NSPrintPanel for that operation.

Short of subclassing NSPrintPanel, you can augment its display by adding a custom NSView through the **setAccessoryView:** method. The panel is automatically resized to accommodate the NSView that you add. Note, however, that you don't have to create controls for special printer features. If a printer includes features in the "OpenUI" field of its PostScript Printer Description (PPD) table, these features will be displayed in a separate panel that's brought up when the user clicks the Print panel's Options button. For more information on a printer's PPD table, see the NSPrinter class description.

Typically, you access an NSPrintPanel by invoking the **printPanel** method. When the class receives a **printPanel** message, it tries to reuse an existing panel rather than create a new one. When a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because a Print panel may be reused, you shouldn't modify the instance returned by **printPanel**, except through the methods listed below. For example, you can set the accessory view, but not the arrangement of the buttons within the panel. If you must modify the Print panel substantially, create and manage your own instance using the **alloc...** and **init...** methods rather than the **printPanel** method.

An application stores printing information in an NSPrintInfo object. NSPrintPanel's **updateFromPrintInfo** reads the NSPrintInfo object's information into the Print panel. **finalWritePrintInfo** updates the NSPrintInfo object if the user changes the information on the Print panel. When the NSPrintOperation object is created, an NSPrintInfo object is also selected for the operation. The NSPrintOperation creates a copy of the NSPrintInfo. **finalWritePrintInfo** actually writes to that copy.

Creating an NSPrintPanel

+ (NSPrintPanel *)**printPanel**	Returns a default NSPrintPanel object.

Customizing the Panel

– (void)**setAccessoryView:**(NSView *)*aView*	Adds an NSView to the panel.
– (NSView *)**accessoryView**	Returns the accessory NSView.

Running the Panel

– (int)**runModal**	Displays the Print panel and begins its event loop. If it is necessary to resize the panel in order to accommodate the list of printers, this method posts the notification NSWindowDidResizeNotification with the receiving object to the default notification center.
– (void)**pickedButton:**(id)*sender*	Stops the event loop.

Updating the Panel's Display

– (void)**pickedAllPages:**(id)*sender*	Updates the panel when the user chooses all pages.
– (void)**pickedLayoutList:**(id)*sender*	Updates the panel when the user chooses a new layout.

Communicating with the NSPrintInfo Object

– (void)**updateFromPrintInfo**	Reads NSPrintPanel's values from the NSPrintInfo object.
– (void)**finalWritePrintInfo**	Writes NSPrintPanel's values to the NSPrintInfo object.

NSResponder

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: AppKit/NSResponder.h

Class Description

NSResponder is an abstract class that forms the basis of command and event processing in the Application Kit. Most Application Kit classes inherit from NSResponder. When an NSResponder receives an event or action message that it can't respond to—that it doesn't have a method for—the message is sent to its *next responder*. For an NSView, the next responder is usually its superview; the content view's next responder is the NSWindow. Each NSWindow, therefore, has its own *responder chain*. Messages are passed up the chain until they reach an object that can respond.

Action messages and keyboard event messages are sent first to the *first responder*, the object that displays the current selection and is expected to handle most user actions within a window. Each NSWindow has its own first responder. Messages the first responder can't handle work their way up the responder chain.

This class defines the methods that pass event and action messages along the responder chain.

Managing the Next Responder

– (NSResponder *)**nextResponder** — Returns the receiver's next responder.

– (void)**setNextResponder:**(NSResponder *)*aResponder*
Makes *aResponder* the receiver's next responder.

Determining the First Responder

– (BOOL)**acceptsFirstResponder** — Subclasses override to accept or reject first responder status. NSResponder's implementation simply returns NO.

– (BOOL)**becomeFirstResponder** — Notifies the receiver that it's the first responder.

– (BOOL)**resignFirstResponder** — Notifies the receiver that it's not the first responder.

Aiding Event Processing

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

Subclasses override to respond to keyboard input. NSResponder's implementation simply returns NO to indicate *theEvent* isn't handled.

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

Aids in dispatching action messages. Returns YES if an responder in the responder chain can perform the *anAction* method, which takes the single argument *anObject*.

Forwarding Event Messages

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

Subclasses override to handle flags-changed events. NSResponder's implementation passes the message to the receiver's next responder.

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

Causes the Help panel to display the help attached to the receiver. If there's no attached help, passes the message to the receiver's next responder.

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

Subclasses override to handle key-down events. NSResponder's implementation passes the message to the receiver's next responder. If the first responder changes, this method posts the notification NSTextDidEndEditingNotification with the current object and, in the notification's dictionary, the key NSTextMovement to the default notification center.

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

Subclasses override to handle key-up events. NSResponder's implementation passes the message to the receiver's next responder.

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

Subclasses override to handle mouse-down events. NSResponder's implementation passes the message to the receiver's next responder.

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

Subclasses override to handle mouse-dragged events. NSResponder's implementation passes the message to the receiver's next responder.

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

Subclasses override to handle mouse-entered events. NSResponder's implementation passes the message to the receiver's next responder.

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

Subclasses override to handle mouse-exited events. NSResponder's implementation passes the message to the receiver's next responder.

– (void)**mouseMoved:**(NSEvent *)*theEvent* Subclasses override to handle mouse-moved events. NSResponder's implementation passes the message to the receiver's next responder.

– (void)**mouseUp:**(NSEvent *)*theEvent* Subclasses override to handle mouse-up events. NSResponder's implementation passes the message to the receiver's next responder.

– (void)**noResponderFor:**(SEL)*eventSelector* Responds to an event message that has reached the end of the responder chain without finding an object that can respond. When the event is a key down, generates a beep.

– (void)**rightMouseDown:**(NSEvent *)*theEvent* Subclasses override to handle right mouse-down events. NSResponder's implementation passes the message to the receiver's next responder.

– (void)**rightMouseDragged:**(NSEvent *)*theEvent* Subclasses override to handle right mouse-dragged events. NSResponder's implementation passes the message to the receiver's next responder.

– (void)**rightMouseUp:**(NSEvent *)*theEvent* Subclasses override to handle right mouse-up events. NSResponder's implementation passes the message to the receiver's next responder.

Services Menu Support

– (id)**validRequestorForSendType:**(NSString *)*typeSent*
returnType:(NSString *)*typeReturned* Subclasses override to determine which Services menu items are enabled at a given time. Returning **self** enables services that can receive *typeSent* pasteboard types and can return *typeReturned* pasteboard types. Returning **nil** disables them. NSResponder's implementation passes the message to the receiver's next responder.

NSSavePanel

Inherits From: NSPanel : NSWindow : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSSavePanel.h

Class Description

NSSavePanel creates a Save panel. The Save panel provides a simple way for a user to specify a file to use when saving a document or other data. It can restrict the user to files of a certain type, as specified by a file name extension.

When the user decides on a file name, the message **panel:isValidFilename:** is sent to the NSSavePanel's delegate (if it responds to that message). The delegate can then determine whether that file name can be used; it returns YES if the file name is valid, or NO if the Save panel should stay up and wait for the user to type in a different file name.

Typically, you access an NSSavePanel by invoking the **savePanel** method. When the class receives a **savePanel** message, it tries to reuse an existing panel rather than create a new one. When a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because a Save panel may be reused, you shouldn't modify the instance returned by **savePanel**, except through the methods listed below. For example, you can set the panel's title and required file type, but not the arrangement of the buttons within the panel. If you must modify the Save panel substantially, create and manage your own instance using the **alloc...** and **init...** methods rather than the **savePanel** method.

Creating an NSSavePanel

+(NSSavePanel *)**savePanel**	Returns an NSSavePanel object, creating it if necessary.

Customizing the NSSavePanel

– (void)**setAccessoryView:**(NSView *)*aView*	Adds an application-customized view to the save panel.
– (NSView *)**accessoryView**	Returns the application-customized view object.
– (void)**setTitle:**(NSString *)*title*	Sets the title of the NSSavePanel to *title*.
– (NSString *)**title**	Returns the title of the NSSavePanel.
– (void)**setPrompt:**(NSString *)*prompt*	Sets the title of the form field for the path to *prompt*.
– (NSString *)**prompt**	Returns the title of the form field for the path.

Setting Directory and File Type

– (NSString *)**requiredFileType** Gets the required file type (if any).

– (void)**setDirectory:**(NSString *)*path* Sets the current directory of the NSSavePanel.

– (void)**setRequiredFileType:**(NSString *)*type* Sets the required file type (if any). An empty string indicates that the user can save to any ASCII file.

– (void)**setTreatsFilePackagesAsDirectories:**(BOOL)*flag* Sets whether the NSSavePanel object treats file packages as directories by showing their contents in the browser.

– (BOOL)**treatsFilePackagesAsDirectories** Returns YES if the NSSavePanel treats file packages as directories, thereby allowing users to browse the contents of file packages.

Running the NSSavePanel

– (int)**runModalForDirectory:**(NSString *)*path* **file:**(NSString *)*filename* Displays the NSSavePanel and begins its event loop, showing *path* in the browser and selecting *filename*.

– (int)**runModal** Displays the NSSavePanel and begins its event loop.

Reading Save Information

– (NSString *)**directory** Returns the directory that the chosen file resides in.

– (NSString *)**filename** Returns the absolute path name of the file to be saved.

Target and Action Methods

– (void)**ok:**(id)*sender* Method invoked by the OK button.

– (void)**cancel:**(id)*sender* Method invoked by the Cancel button.

Responding to User Input

– (void)**selectText:**(id)*sender* Invoked when users press Tab, Shift-Tab, or an arrow key.

Setting the Delegate

– (void)**setDelegate:**(id)*anObject* Makes *anObject* the NSSavePanel's delegate.

Methods Implemented by the Delegate

– (NSComparisonResult)**panel:**(id)*sender*
compareFilename:(NSString *)*filename1*
with:(NSString *)*filename2*
caseSensitive:(BOOL)*caseSensitive*

Returns NSOrderedDescending if *filename1* precedes *filename2*, NSOrderedAscending in the opposite case, NSOrderedSame if the two are equivalent.

– (BOOL)**panel:**(id)*sender*
shouldShowFilename:(NSString *)*filename*

Returns YES if *filename* should be displayed in the browser.

– (BOOL)**panel:**(id)*sender*
isValidFilename:(NSString*)*filename*

Returns YES if *filename* is acceptable to the delegate.

NSScreen

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSScreen.h

Class Description

An NSScreen object describes the attributes of a computer's monitor, or screen. An application may use an NSScreen object to retrieve information about a screen and use this information to decide what to display upon that screen. For example, an application may use the **deepestScreen** method to find out which of the available screens can best represent color and then may choose to display all of its windows on that screen.

The two main attributes of a screen are its depth and its dimensions. The **depth** method describes the screen depth (such as two-bit, eight-bit, or twelve-bit) and tells you if the screen can display color. The **frame** method gives the screen's dimensions and location as an NSRect.

The device description dictionary contains more complete information about the screen. Use NSScreen's **deviceDescription** method to access the dictionary, and use these keys to retrieve information about a screen:

Dictionary Key	Returns
NSDeviceResolution	An NSValue describing the screen's resolution in dots per inch (dpi).
NSDeviceColorSpaceName	The screen's color space name. See NSGraphics.h for a list of possible values.
NSDeviceBitsPerSample	The bit depth of screen images (2-bit, 8-bit, etc.).
NSDeviceIsScreen	YES, indicating the device is a screen.
NSDeviceSize	An NSValue describing the screen's size in points.

The device description dictionary contains information about not only screens, but all other system devices such as printers and windows. There are other keys into the dictionary that you would use to obtain information about these other devices. For a complete list of device dictionary keys, see NSGraphics.h.

Creating NSScreen Instances

+ (NSScreen *)**mainScreen**

Returns an NSScreen object representing the main screen. The main screen is the screen with the key window.

+ (NSScreen *)**deepestScreen**

Returns an NSScreen object representing the screen that can best represent color. This method always returns an object, even if there is only one screen and it is not a color screen.

+ (NSArray *)**screens**

Returns an array of NSScreen objects representing all of the screens available on the system. Raises NSWindowServerCommunicationException if the screens information can't be obtained from the window system.

Reading Screen Information

– (NSWindowDepth)**depth**

Returns the screen's depth, including whether the screen can display color.

– (NSRect)**frame**

Returns the dimensions and location of the screen in an NSRect.

– (NSDictionary *)**deviceDescription**

Returns the device dictionary as described in the class description.

NSScroller

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSScroller.h

Class Description

The NSScroller class defines a control that's used by an NSScrollView object to position a document that's too large to be displayed in its entirety within an NSView. An NSScroller is typically represented on the screen by a bar, a knob, and two scroll buttons, although it may contain only some of these. The knob indicates both the position within the document and the amount displayed relative to the size of the document. The bar is the rectangular region that the knob slides within. The scroll buttons allow the user to scroll in small increments by clicking, or in large increments by Alternate-clicking. In discussions of the NSScroller class, a small increment is referred to as a "line increment" (even if the NSScroller is oriented horizontally), and a large increment is referred to as a "page increment," although a page increment actually advances the document by one windowful. When you create an NSScroller, you can specify either a vertical or a horizontal orientation.

As an NSControl, an NSScroller handles mouse events and sends action messages to its target (usually its parent NSScrollView) to implement user-controlled scrolling. The NSScroller must also respond to messages from an NSScrollView to represent changes in document positioning.

NSScroller is a public class primarily for programmers who decide not to use an NSScrollView but want to present a consistent user interface. Its use is not encouraged except in cases where the porting of an existing application is made more straightforward. In these situations, you initialize a newly created NSScroller by calling **initWithFrame:**. Then, you use **setTarget:** (NSControl) to set the object that will receive messages from the NSScroller, and you use **setAction:** (NSControl) to specify the message that will be sent to the target by the NSScroller. When your target receives a message from the NSScroller, it will probably need to query the NSScroller using the **hitPart** and **floatValue** (NSControl) methods to determine what action to take.

The NSScroller class has several constants referring to the parts of an NSScroller. A scroll button with an up arrow (or left arrow, if the NSScroller is oriented horizontally) is known as a "decrement line" button if it receives a normal click, and as a "decrement page" button if it receives an Alternate-click. Similarly, a scroll button with a down or right arrow functions as both an "increment line" button and an "increment page" button. The constants defining the parts of an NSScroller are as follows:

Constant	Refers To
NSScrollerNoPart	No part of the NSScroller
NSScrollerKnob	The knob
NSScrollerDecrementPage	The button that decrements a windowful (up or left arrow)
NSScrollerIncrementPage	The button that increments a windowful (down or right arrow)
NSScrollerDecrementLine	The button that decrements a windowful (up or left arrow)
NSScrollerIncrementLine	The button that increments a windowful (down or right arrow)
NSScrollerKnobSlot	The bar

The following constants are used in the **setArrowsPosition:** method to set the position of the scroll buttons within the scroller:

Constant	Meaning
NSScrollerArrowsMaxEnd	Scroll buttons are placed at the bottom or right end of the scroller.
NSScrollerArrowsMinEnd	Scroll buttons are placed at the top or left part of the scroller.
NSScrollerArrowsNone	The scroller doesn't have scroll buttons.

An NSScroller can be made too small for all its parts to be displayed. The **usableParts** method returns one of the following constants to indicate whether such a condition is present:

Constant	Meaning
NSNoScrollerParts	Scroller has no usable parts, only the bar.
NSOnlyScrollerArrows	Scroller has only scroll buttons.
NSAllScrollerParts	Scroller has all parts.

The following constants are used as values for the first argument of the **drawArrow:highlight:** method, to indicate which scroll button is to be drawn:

Constant	Meaning
NSScrollerIncrementArrow	The scroll button that scrolls forward.
NSScrollerDecrementArrow	The scroll button that scrolls backward.

Laying out the NSScroller

+ (float)**scrollerWidth** — Returns the width of the scoller, a constant value.

– (NSScrollArrowPosition)**arrowsPosition** — Returns the position of scroll arrows in the NSScroller.

– (void)**checkSpaceForParts** — Checks for room for knob and scroll buttons.

– (NSRect)**rectForPart:**(NSScrollerPart)*partCode* — Gets the rectangle that encloses *partCode*.

– (void)**setArrowsPosition:**(NSScrollArrowPosition)*where* — Sets position of scroll arrows in the NSScroller.

– (NSUsableScrollerParts)**usableParts** — Indicates which parts of the scroller can be displayed, given the NSScroller's current size.

Setting the NSScroller's Values

– (float)**knobProportion**	Returns the ratio of the knob's length to the NSScroller's length.
– (void)**setFloatValue:**(float)*aFloat* **knobProportion:**(float)*ratio*	Sets the NSScroller's value, repositioning the knob according to *aFloat* and resizing it according to *ratio*. Both arguments are clipped to the range from 0.0 to 1.0, inclusive.

Displaying

– (void)**drawArrow:**(NSScrollerArrow)*whichButton* **highlight:**(BOOL)*flag*	Draws highlighted and unhighlighted arrows.
– (void)**drawKnob**	Draws the knob.
– (void)**drawParts**	Caches bitmaps for knob and scroll arrows.
– (void)**highlight:**(BOOL)*flag*	Highlights scroll button that's under mouse.

Handling Events

– (NSScrollerPart)**hitPart**	Returns the part of the NSScroller object that received mouse-down.
– (NSScrollerPart)**testPart:**(NSPoint)*thePoint*	Returns the part of the NSScroller that's under *thePoint*.
– (void)**trackKnob:**(NSEvent *)*theEvent*	Invoked in response to mouse-down events on the knob.
– (void)**trackScrollButtons:**(NSEvent *)*theEvent*	Invoked in response to mouse-down events on buttons.

NSScrollView

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSScrollView.h

Class Description

An NSScrollView object lets the user interact with a document that's too large to be shown in its entirety within an NSView and must therefore be scrolled. The responsibility of an NSScrollView is to coordinate scrolling behavior between NSScroller objects and a NSClipView object. Thus, the user may drag the knob of an NSScroller and the NSScrollView will send a message to its NSClipView to ensure that the viewed portion of the document reflects the position of the knob. Similarly, the application can change the viewed position within a document and the NSScrollView will send a message to the NSScrollers advising them of this change.

The NSScrollView has at least one subview (an NSClipView object), which is called the *content view*. The content view in turn has a subview called the *document view*, which is the view to be scrolled. When an NSScrollView is created, it has neither a vertical nor a horizontal scroller. If NSScrollers are required, the application must send **setHasHorizontalScroller:YES** and **setHasVerticalScroller:YES** messages to the NSScrollView; the content view is resized to fill the area of the NSScrollView not occupied by the NSScrollers.

When the application modifies the scroll position within the document, it should send a **reflectScrolledClipView:** message to the NSScrollView, which will then query the content view and set the NSScroller(s) accordingly. The **reflectScrolledClipView:** message may also cause the NSScrollView to enable or disable the NSScrollers as required.

Determining Component Sizes

– (NSSize)**contentSize**	Gets the content view's size.
– (NSRect)**documentVisibleRect**	Gets the visible portion of the document view.

Laying Out the NSScrollView

+ (NSSize)**contentSizeForFrameSize:**(NSSize)*size* **hasHorizontalScroller:**(BOOL)*horizFlag* **hasVerticalScroller:**(BOOL)*vertFlag* **borderType:**(NSBorderType)*aType*	Gets the content view size for the given NSScrollView frame size.
+ (NSSize)**frameSizeForContentSize:**(NSSize)*size* **hasHorizontalScroller:**(BOOL)*horizFlag* **hasVerticalScroller:**(BOOL)*vertFlag* **borderType:**(NSBorderType)*aType*	Gets the NSScrollView frame size for the given content view size.
– (void)**setHasHorizontalScroller:**(BOOL)*flag*	Instructs the NSScrollView whether to create and use a horizontal scroller.
– (BOOL)**hasHorizontalScroller**	Returns YES if the NSScrollView object has a horizontal scroller.
– (void)**setHasVerticalScroller:**(BOOL)*flag*	Instructs the NSScrollView whether to create and use a vertical scroller.
– (BOOL)**hasVerticalScroller**	Returns YES if the NSScrollView object has a vertical scroller.
– (void)**tile**	Retiles the scrollers and content view.
– (void)**toggleRuler:**(id)*sender*	Makes the ruler visible or invisible, whichever is the opposite of its current state.
– (BOOL)**isRulerVisible**	Returns whether the ruler is visible in the NSScrollView.

Managing Component Views

– (void)**setDocumentView:**(NSView *)*aView*	Makes *aView* the NSScrollView's document view.
– (id)**documentView**	Returns the current document view.
– (void)**setHorizontalScroller:**(NSScroller *)*anObject*	Sets the horizontal NSScroller object.
– (NSScroller *)**horizontalScroller**	Returns the horizontal NSScroller object.
– (void)**setVerticalScroller:**(NSScroller *)*anObject*	Sets the vertical NSScroller object.
– (NSScroller *)**verticalScroller**	Returns the vertical NSScroller object.
– (void)**reflectScrolledClipView:**(NSClipView *)*cView*	Moves the scrollers to reflect change in the coordinates of the clip view.

Modifying Graphic Attributes

– (void)**setBorderType:**(NSBorderType)*aType*	Sets the border type of the NSScrollView.
– (NSBorderType)**borderType**	Returns the border type.
– (void)**setBackgroundColor:**(NSColor *)*color*	Sets the NSScrollView's background color.
– (NSColor *)**backgroundColor**	Returns the NSScrollView's background color.

Setting Scrolling Behavior

– (float)**lineScroll**	Returns the amount scrolled when scrolling a line. (The return value is expressed in units of the NSScrollView's coordinate system.)
– (float)**pageScroll**	Returns the amount scrolled when scrolling a page. (The return value is expressed in units of the NSScrollView's coordinate system.)
– (void)**setScrollsDynamically:**(BOOL)*flag*	Sets how the document view is displayed during scrolling.
– (BOOL)**scrollsDynamically**	Returns whether the NSScrollView scrolls dynamically.
– (void)**setLineScroll:**(float)*value*	Sets the amount to scroll when scrolling a line.
– (void)**setPageScroll:**(float)*value*	Sets the amount of overlap for a page scroll.

Managing the Cursor

– (void)**setDocumentCursor:**(NSCursor *)*anObject*	Sets the cursor for the document view.

NSSelection

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: AppKit/NSSelection.h

Class Description

The NSSelection class defines an object that describes a selection within a document. An NSSelection, or simply, selection, is an immutable description; it may be held by the system or other documents, and it cannot change over time. Selections are typically used by NSDataLink objects to represent the source and destination of a link.

Because a selection description can't be changed once it's been exported, it's a good idea to construct general descriptions that can survive changes to a document and don't require selection-specific information to be stored in the document. This description may be simple or complex, depending upon the application. For example, a painting application might describe a selection in an image as a simple rectangle. This description doesn't require that any information be stored in the image's file, and the description can be expected to remain valid through the life of the image. An object-based drawing application might describe a selection as a list of object identifiers (though *not* **id**s), where an object identifier is unique throughout the life of the document. Based on this list, a selection could be meaningfully reconstructed, even if new objects are added to the document or selected objects are deleted. Such a scheme doesn't require that any selection-specific information be stored in the document's file, with the benefit that links can be made to read-only documents.

Maintaining a character-range selection in a text document is more problematic. A possible solution is to insert selection-begin and selection-end markers that define a specific selection into the text stream. A selection description would then refer to a specific selection marker. This solution requires that selection state information be stored and maintained within the document. Furthermore, this information generally shouldn't be purged from the document, because the document can't know how many references to the selection exist. (References to the selection could be stored with documents on removable media, like floppy disks.) This selection-state information should be maintained as long as it refers to any meaningful data. For this reason, it's desirable to describe selection in a manner that doesn't require that selection-state information be maintained in the document whenever possible.

Three well-known selection descriptions can apply to any document: the empty selection, the entire document, and the abstract concept of the current selection. NSSelection objects for these selections are returned by the **emptySelection**, **allSelection**, and **currentSelection** class methods.

Since an NSSelection may be used in a document that is read by machines with different architectures, care should be taken to write machine-independent descriptions. For example, using a binary structure as a selection description will fail on a machine where an identically defined structure has a different size or is kept in memory with different byte ordering. Exporting (and then parsing) ASCII descriptions is often a good solution. If binary descriptions must be used, it's prudent to preface the description with a token specifying the description's byte ordering.

It may also be prudent to version-stamp selection descriptions, so that old selections can be accurately read by updated versions of an application.

Returning Special Selection Shared Instances

+ (NSSelection *)**allSelection**
Returns the shared instance of the well-known selection representing the entire document.

+ (NSSelection *)**currentSelection**
Returns the shared instance of the well-known selection representing the abstract concept of the current selection. The current selection never describes a specific selection; it describes a selection that may change frequently.

+ (NSSelection *)**emptySelection**
Returns the shared instance of the well-known selection representing no data.

Creating and Initializing a Selection

+ (NSSelection *)**selectionWithDescriptionData:**(NSData *)*data*
Creates and returns an NSSelection object that records *data* as the description of the selection.

– (id)**initWithDescriptionData:**(NSData *)*newData*
Initializes a newly allocated NSSelection object that records *data* as the description of the selection. Returns the initialized object.

– (id)**initWithPasteboard:**(NSPasteboard *)*pasteboard*
Initializes a newly allocated NSSelection object that takes its description of the selection from *pasteboard*. Returns the initialized object.

Describing a Selection

– (NSData *)**descriptionData**
Returns the data that describes the selection as set by **selectionWithDescriptionData:** or **initWithDescriptionData:**.

– (BOOL)**isWellKnownSelection**
Returns YES if the receiver is one of the well-known selection types (those representing the entire document, current selection, or empty selection) and NO otherwise.

Writing a Selection to the Pasteboard

– (void)**writeToPasteboard:**(NSPasteboard *)*pasteboard*

Writes the selection data to the pasteboard *pasteboard*. A copy of the selection can then be retrieved by initializing a new NSSelection from the pasteboard using **initWithPasteboard:**.

NSSlider

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSSlider.h

Class Description

NSSlider is a type of NSControl with a sliding knob that can be moved to represent a value between a minimum and a maximum setting. A slider may be either horizontal or vertical, but its minimum value is always at the left or bottom end of the bar, and the maximum at the right or top. By default, an NSSlider is a continuous NSControl: It sends its action message to its target continuously while the user drags its knob. To configure an NSSlider to send its action only when the mouse is released, send **setContinuous:** (an NSControl method) with an argument of NO.

An NSSlider can be configured to display an image, a title, or both, in the area behind its knob. An NSSlider's title can be drawn in any gray level or color, and in any font available. An NSSlider's value can be set programmatically with any of the standard NSControl value-setting methods, such as **setFloatValue:**.

For more information, see the method descriptions in the NSSliderCell class specification.

Setting the Cell Class

+ (Class)**cellClass**	Returns the class last set in a **setCellClass:** message, or the NSSliderCell class if **setCellClass:** has never been called.
+ (void)**setCellClass:**(Class)*classId*	Sets the class of NSCell used in the NSSlider.

Modifying an NSSlider's Appearance

– (NSImage *)**image**	Returns the image within the NSSlider.
– (int)**isVertical**	Returns 1 if the NSSlider is vertical, 0 if horizontal, -1 if unknown.
– (float)**knobThickness**	Returns the knob's thickness as a float value (width if horizontal slider, height if vertical slider).
– (void)**setImage:**(NSImage *)*backgroundImage*	Sets the image within the NSSlider to *backgroundImage*.

– (void)**setKnobThickness:**(float)*aFloat* — Sets the knob's thickness (its width if the slider is horizontal, height if vertical) to *aFloat*, expressed in units of the NSSlider's coordinate system.

– (void)**setTitle:**(NSString *)*aString* — Sets the title within the NSSlider to a copy of *aString*.

– (void)**setTitleCell:**(NSCell *)*aCell* — Sets the NSCell (or subclass thereof) object used to draw the title within the NSSlider. The cell object should ideally be an instance of NSTextFieldCell or one of its subclasses.

– (void)**setTitleColor:**(NSColor *)*aColor* — Sets the color of text in the title to *aColor.*

– (void)**setTitleFont:**(NSFont *)*fontObject* — Sets the NSFont object used for the title within the NSSlider.

– (NSString *)**title** — Returns the title within the NSSlider.

– (id)**titleCell** — Returns the NSCell (or subclass thereof) object used to draw the title within the NSSlider.

– (NSColor *)**titleColor** — Returns the color of text in the title.

– (NSFont *)**titleFont** — Returns the NSFont object used in drawing the title within the NSSlider.

Setting and Getting V alue Limits

– (double)**maxValue** — Returns theNSSlider's maximum value.

– (double)**minValue** — Returns the NSSlider's minimum value.

– (void)**setMaxValue:**(double)*aDouble* — Sets the NSSlider's maximum value to *aDouble*.

– (void)**setMinValue:**(double)*aDouble* — Sets the NSSlider's minimum value to *aDouble*.

Handling Events

– (BOOL)**acceptsFirstMouse:**(NSEvent *)*theEvent* — Returns YES by default, since NSSliders always accept a mouse-down event that activates a window, whether or not the NSSlider is enabled. Override this if you want different behavior.

NSSliderCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSSliderCell.h

Class Description

NSSliderCell is a type of NSCell used to assist the NSSlider class, and to build matrices of sliders. The NSSliderCell encompasses all the visible portions of the NSSlider—the knob, the area along which the knob slides, and the optional title within this area. See the NSSlider class specification for an overview of how NSSliderCells work.

Determining Component Sizes

– (NSSize)**cellSizeForBounds:**(NSRect)*aRect*
Returns the minimum width and height needed to draw the NSSliderCell in *aRect*. If *aRect* too small to fit the knob and bezel, the width and height of *theSize* are set to 0.0.

– (NSRect)**knobRectFlipped:**(BOOL)*flipped*
Gets the rectangle the knob will be drawn in. *flipped* indicates whether the NSSliderCell's view has a flipped coordinate system.

Setting Value Limits

– (double)**maxValue**
Returns the NSSliderCell’s maximum value.

– (double)**minValue**
Returns the NSSliderCell’s minimum value.

– (void)**setMaxValue:**(double)*aDouble*
Sets the maximum value of the NSSliderCell to *aDouble*.

– (void)**setMinValue:**(double)*aDouble*
Sets the NSSliderCell’s minimum value to *aDouble*.

Modifying Graphic Attributes

– (int)**isVertical**
Returns 1 if the NSSliderCell is vertical, 0 if horizontal, -1 if unknown.

– (float)**knobThickness**
Returns the knob’s thickness as a float value.

– (void)**setKnobThickness:**(float)*aFloat*
Sets the knob’s thickness to *aFloat* (width if a horizontal slider, height if vertical).

– (void)**setTitle:**(NSString *)*aString* — Sets the title within the NSSliderCell to a copy of *aString*.

– (void)**setTitleCell:**(NSCell *)*aCell* — Sets the NSCell (or subclass thereof) object used to draw the title within the NSSliderCell. The cell object should ideally be an instance of NSTextFieldCell or one of its subclasses.

– (void)**setTitleColor:**(NSColor *)*aColor* — Sets the color of text in the title to *aColor.*

– (void)**setTitleFont:**(NSFont *)*fontObject* — Sets the NSFont object used to draw the title within the NSSliderCell.

– (NSString *)**title** — Returns the title within the NSSliderCell.

– (id)**titleCell** — Returns the NSCell (or subclass thereof) object used to draw the title within the NSSliderCell.

– (NSFont *)**titleFont** — Returns the NSFont object used in drawing the title within the NSSliderCell.

– (NSColor *)**titleColor** — Returns the color of text in the title.

Displaying the NSSliderCell

– (void)**drawBarInside:**(NSRect)*aRect* **flipped:**(BOOL)*flipped* — Draws the NSSliderCell's background bar (but not the bezel around it or the knob) in *aRect*. *flipped* indicates whether the NSView's coordinate system is flipped.

– (void)**drawKnob** — Draws the NSSliderCell's knob after calculating the drawing rectangle.

– (void)**drawKnob:**(NSRect)*knobRect* — Draws the NSSliderCell's knob in *knobRect*.

Modifying Behavior

– (double)**altIncrementValue** — Returns the increment by which the NSSliderCell modifies its value when its knob is Alternate-dragged one pixel.

– (void)**setAltIncrementValue:**(double)*incValue* — Sets the amount by which the NSSliderCell modifies its value when the knob is dragged one pixel with the Alternate key held down.

Tracking the Mouse

+ (BOOL)**prefersTrackingUntilMouseUp**

Returns YES to allow NSSliderCell objects to track even when the mouse leaves their bounds. Override this method to return NO if you want the NSSliderCell to stop tracking once the mouse leaves its bounds.

– (NSRect)**trackRect**

Returns the rectangle used in tracking the mouse (only valid while tracking).

NSSpellChecker

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSSpellChecker.h

Class Description

The NSSpellChecker class gives any application an interface to the OpenStep spell-checking service. To handle all its spell checking, an application needs only one instance of NSSpellChecker. It provides a panel in which the user can specify decisions about words that are suspect. To check the spelling of a piece of text, the application:

- Includes in its user interface a menu item (or a button or command) by which the user will request spell checking.
- Makes the text available by way of an NSString object.
- Creates an instance of the NSSpellChecker class and sends it a **checkSpellingOfString:startingAt:** message.

For example, you might use the following statement to create an NSSpellChecker:

```
range = [[NSSpellChecker sharedSpellChecker] checkSpellingOfString:aString startingAt:0];
```

The **checkSpellingOfString:startingAt:** method checks the spelling of the words in the specified string beginning at the specified offset (this example uses 0 to start at the beginning of the string) until it finds a word that is misspelled. Then it returns an NSRange to indicate the location of the misspelled word.

In a graphical application, whenever a misspelled word is found, you'll probably want to highlight the word in the document, using the NSRange that **checkSpellingOfString:startingAt:** returned to determine the text to highlight. Then you should show the misspelled word in the Spelling panel's misspelled-word field by calling **updateSpellingPanelWithMisspelledWord:**. If **checkSpellingOfString:startingAt:** does not find a misspelled word, you should call **updateSpellingPanelWithMisspelledWord:** with the empty string. This causes the system to beep, letting the user know that the spell check is complete and no misspelled words were found. None of these steps is required, but if you do one, you should do them all.

The object that provides the string being checked should adopt the following protocols:

NSChangeSpelling	A message in this protocol (**changeSpelling:**) is sent down the responder chain when the user presses the Correct button.
NSIgnoreMisspelledWords	When the object being checked responds to this protocol, the spell server keeps a list of words that are acceptable in the document and enables the Ignore button in the Spelling panel.

The application may choose to split a document's text into segments and check them separately. This will be necessary when the text has segments in different languages. Spell checking is invoked for one language at a time, so a document that contains portions in three languages will require at least three checks.

Dictionaries and Word Lists

The process of checking spelling makes use of three references:

- A dictionary registered with the system's spell-checking service. When the Spelling panel first appears, by default it shows the dictionary for the user's preferred language. The user may select a different dictionary from the list in the Spelling panel.
- The user's "learn" list of correctly-spelled words in the current language. The NSSpellChecker updates the list when the user presses the Learn or Forget buttons in the Spelling panel.
- The document's list of words to be ignored while checking it (if the first responder conforms to the NSIgnoreMisspelledWords protocol). The NSSpellChecker updates its copy of this list when the user presses the Ignore button in the Spelling panel.

A word is considered to be misspelled if none of these three accepts it.

Matching a List of Ignored Words with the Document It Belongs To

The NSString being checked isn't the same as the document. In the course of processing a document, an application might run several checks based on different parts or different versions of the text. But they'd all belong to the same document. The NSSpellChecker keeps a separate "ignored words" list for each document that it checks. To help match "ignored words" lists to documents, you should call **uniqueSpellDocumentTag** once for each document. This method returns a unique arbitrary integer that will serve to distinguish one document from the others being checked and to match each "ignored words" list to a document. When searching for misspelled words, pass the tag as the fourth argument of **checkSpellingOfString:startingAt:language:wrap:inSpellDocumentWithTag:wordCount:**. (The convenience method **checkSpellingOfString:startingAt:** takes no tag. This method is suitable when the first responder does not conform to the NSIgnoreMisspelledWords protocol.)

When the application saves a document, it may choose to retrieve the "ignored words" list and save it along with the document. To get back the right list, it must send the NSSpellChecker an **ignoredWordsInSpellDocumentWithTag:** message. When the application has closed a document, it should notify the NSSpellChecker that the document's "ignored words" list can now be discarded, by sending it a **closeSpellDocumentWithTag:** message. When the application reopens the document, it should restore the "ignored words" list with the message **setIgnoredWords:inSpellDocumentWithTag:**.

Making a Checker available

+ (NSSpellChecker *)**sharedSpellChecker**	Returns the NSSpellChecker (one per application).
+ (BOOL)**sharedSpellCheckerExists**	Returns whether the application's NSSpellChecker has already been created.

Managing the Spelling Panel

– (NSView *)**accessoryView**

Returns the Spelling panel's accessory NSView object.

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

Makes an NSView object an accessory of the Spelling panel by making it a subview of the panel's content view. This method posts the notification NSWindowDidResizeNotification with the Spelling panel object to the default notification center.

– (NSPanel *)**spellingPanel**

Returns the NSSpellChecker's panel.

Checking Spelling

– (int)**countWordsInString:**(NSString *)*aString*
language:(NSString *)*language*

Returns the number of words in *string*. The *language* argument specifies the language used in the string. If *language* is the empty string, the current selection in the Spelling panel's pop-up menu is used.

– (NSRange)**checkSpellingOfString:**(NSString *)*stringToCheck*
startingAt:(int)*startingOffset*

Starts the search for a misspelled word in *stringToCheck* starting at *startingOffset* within the string object. Returns the range of the first misspelled word. Wrapping occurs but no ignored-words dictionary is used.

– (NSRange)**checkSpellingOfString:**(NSString *)*stringToCheck*
startingAt:(int)*startingOffset*
language:(NSString *)*language*
wrap:(BOOL)*wrapFlag*
inSpellDocumentWithTag:(int)*tag*
wordCount:(int *)*wordCount*

Starts the search for a misspelled word in *stringToCheck* starting at *startingOffset* within the string object. Returns the range of the first misspelled word and optionally the word count by reference. *tag* is an identifier unique within the application used to inform the spell check which document (actually, a dictionary) of ignored words to use. *wrapFlag* determines whether spell checking continues at the beginning of the string when the end is reached. *language* is the language used in the string. If *language* is the empty string, the current selection in the Spelling panel's pop-up menu is used.

Setting the Language

– (NSString *)**language**

Returns the current language used in spell-checking.

– (BOOL)**setLanguage:**(NSString *)*aLanguage*

Sets the language to use in spell-checking to *aLanguage*. Returns whether the Language pop-up list in the Spelling panel lists *aLanguage*.

Managing the Spelling Process

+ (int)**uniqueSpellDocumentTag**

Returns a guaranteed unique tag to use as the spell-document tag for a document. Use this method to generate tags to avoid collisions with other objects that can be spell-checked.

– (void)**closeSpellDocumentWithTag:**(int)*tag*

Notifies the NSSpellChecker that the user has finished with the ignored-word document identified by *tag*, causing it to throw that dictionary away.

– (void)**ignoreWord:**(NSString *)*wordToIgnore* **inSpellDocumentWithTag:**(int)*tag*

Instructs the NSSpellChecker to ignore all future occurrences of *wordToIgnore* in the document identified by *tag*. You should call this method from within your implementation of the NSIgnoreMisspelledWords protocol's **ignoreSpelling:**.

– (NSArray *)**ignoredWordsInSpellDocumentWithTag:**(int)*tag*

Returns the array of ignored words for a document identified by *tag*. Invoke this before **closeSpellDocument:** if you want to store the ignored words.

– (void)**setIgnoredWords:**(NSArray *)*someWords* **inSpellDocumentWithTag:**(int)*tag*

Initializes the ignored-words document (i.e., dictionary identified by *tag* with *someWords*, an array of words to ignore.

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

Sets the string that appears in the misspelled word field, using the string object *aString*.

– (void)**updateSpellingPanelWithMisspelledWord:**(NSString *)*word*

Causes NSSpellChecker to update the Spelling panel's misspelled-word field to reflect *word*. You are responsible for highlighting *word* in the document and for extracting it from the document using the range returned by the **checkSpelling:...** methods. Pass the empty string as *word* to have the system beep, indicating no misspelled words were found.

NSSpellServer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSSpellServer.h

Class Description

The NSSpellServer class gives you a way to make your particular spelling checker a service that's available to any application. A *service* is an application that declares its availability in a standard way, so that any other applications that wish to use it can do so. If you build a spelling checker that makes use of the NSSpellServer class and list it as an available service, then users of any application that makes use of NSSpellChecker or includes a Services menu will see your spelling checker as one of the available dictionaries.

To make use of NSSpellServer, you write a small program that creates an NSSpellServer instance and a delegate that responds to messages asking it to find a misspelled word and to suggest guesses for a misspelled word. Send the NSSpellServer **registerLanguage:byVendor:** messages to tell it the languages your delegate can handle.

The program that runs your spelling checker should not be built as an Application Kit application, but as a simple program. Suppose you supply spelling checkers under the vendor name "Acme." Suppose the file containing the code for your delegate is called AcmeEnglishSpellChecker. Then the following might be your program's **main**:

```
void main()
{
  NSSpellServer *aServer = [[NSSpellServer alloc] init];
  if ([aServer registerLanguage:@"English" byVendor:@"Acme"]) {
    [aServer setDelegate:[AcmeEnglishSpellChecker alloc] init]];
    [aServer run];
    fprintf(stderr, "Unexpected death of Acme SpellChecker!\n");
  } else {
    fprintf(stderr, "Unable to check in Acme SpellChecker.\n");
  }
}
```

Your delegate is an instance of a custom subclass. (It's simplest to make it a subclass of NSObject, but that's not a requirement.) Given an NSString, your delegate must be able to find a misspelled word by implementing the method **spellServer:findMisspelledWordInString:language:wordCount:countOnly:**. Usually, this method also reports the number of words it has scanned, but that isn't mandatory.

Optionally, the delegate may also suggest corrections for misspelled words. It does so by implementing the method **spellServer:suggestGuessesForWord:inLanguage:**

Service Availability Notice

When there's more than one spelling checker available, the user selects the one desired. The application that requests a spelling check uses an NSSpellChecker object, and it provides a Spelling panel; in the panel there's a pop-up list of available spelling checkers. Your spelling checker appears in that list if it has a *service descriptor.*

A service descriptor is an entry in a text file called **services**. Usually it's located within the bundle that also contains your spelling checker's executable file. The bundle (or directory) that contains the services file must have a name ending in ".service" or ".app". The system looks for service bundles in a standard set of directories.

A spell checker service availability notice has a standard format, illustrated in the following example for the Acme spelling checker:

```
Spell Checker:  Acme
Language:  French
Language:  English
Executable:  franglais.daemon
```

The first line identifies the type of service; for a spelling checker, it must say "Spell Checker:" followed by your vendor name. The next line contains the English name of a language your spelling checker is prepared to check. (The language must be one your system recognizes.) If your program can check more than one language, use an additional line for each additional language. The last line of a descriptor gives the name of the service's executable file. (It requires a complete path if it's in a different directory.)

If there's a service descriptor for your Acme spelling checker and also a service descriptor for the English checker provided by a vendor named Consolidated, a user looking at the Spelling panel's pop-up list would see:

```
English (Acme)
English (Consolidated)
French (Acme)
```

Illustrative Sequence of Messages to an NSSpellServer

The act of checking spelling usually involves the interplay of objects in two classes: the user application's NSSpellChecker (which responds to interactions with the user) and your spelling checker's NSSpellServer (which provides the application interface for your spelling checker). You can see the interaction between the two in the following list of steps involved in finding a misspelled word.

- The user of an application selects a menu item to request a spelling check. The application sends a message to its NSSpellChecker object. The NSSpellChecker in turn sends a corresponding message to the appropriate NSSpellServer.
- The NSSpellServer receives the message asking it to check the spelling of an NSString. It forwards the message to its delegate.
- The delegate searches for a misspelled word. If it finds one, it returns an NSRange identifying the word's location in the string.
- The NSSpellServer receives a message asking it to suggest guesses for the correct spelling of a misspelled word, and forwards the message to its delegate.

- The delegate returns a list of possible corrections, which the NSSpellServer in turn returns to the NSSpellChecker that initiated the request.
- The NSSpellServer doesn't know what the user does with the errors its delegate has found or with the guesses its delegate has proposed. (Perhaps the user corrects the document, perhaps by selecting a correction from the NSSpellChecker's display of guesses; but that's not the NSSpellServer's responsibility.) However, if the user presses the Learn or Forget buttons (thereby causing the NSSpellChecker to revise the user's word list), the NSSpellServer receives a notification of the word thus learned or forgotten. It's up to you whether your spell checker acts on this information. If the user presses the Ignore button, the delegate is not notified (but the next time that word occurs in the text, the method **isWordInUserDictionaries:caseSensitive:** will report YES rather than NO).
- Once the NSSpellServer delegate has reported a misspelled word, it has completed its search. Of course, it's likely that the user's application will then send a new message, this time asking the NSSpellServer to check a string containing the part of the text it didn't get to earlier.

Checking in Your Service

– (BOOL)**registerLanguage:**(NSString *)*language*
byVendor:(NSString *)*vendor*
Registers a spelling server for *language* by *vendor*.

Assigning a Delegate

– (id)**delegate**
Returns the NSSpellServer's delegate.

– (void)**setDelegate:**(id)*anObject*
Sets the delegate of the NSSpellServer.

Running the Service

– (void)**run**
Makes the NSSpellServer start listening for spell-checking requests. This method should not return.

Checking User Dictionaries

– (BOOL)**isWordInUserDictionaries:**(NSString *)*word*
caseSensitive:(BOOL)*flag*
Returns whether *word* is in any open user dictionary; the search is case-sensitive if *flag* is YES.

Methods Implemented by the Delegate

– (NSRange)**spellServer:**(NSSpellServer *)*sender* **findMisspelledWordInString:** (NSString *)*stringToCheck* **language:**(NSString *)*language* **wordCount:**(int *)*wordCount* **countOnly:**(BOOL)*countOnly*	Search for a misspelled word in *stringToCheck*, using *language*, and marking the first misspelled word found by returning its range within the string object. In *wordCount* return by reference the number of words from the beginning of the string object until the misspelled word (or the end-of-string). If *countOnly* is YES, just count the words in the string object; do not spell-check. Send **isWordInUserDictionaries:caseSensitive:** to the spelling server to determine if *word* exists in the user's language dictionaries.
– (NSArray *)**spellServer:**(NSSpellServer *)*sender* **suggestGuessesForWord:**(NSString *)*word* **inLanguage:**(NSString *)*language*	Search for alternatives to the misspelled *word* in *language*. Return guesses as an array of string objects.
– (void)**spellServer:**(NSSpellServer *)*sender* **didLearnWord:**(NSString *)*word* **inLanguage:**(NSString *)*language*	Notifies the delegate of a *word* added to the user's hidden word list.
– (void)**spellServer:**(NSSpellServer *)*sender* **didForgetWord:**(NSString *)*word* **inLanguage:**(NSString *)*language*	Notifies the delegate of a *word* removed from the user's hidden word list.

NSSplitView

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSSplitView.h

Class Description

An NSSplitView object lets several views share a region within a window. The NSSplitView resizes its subviews so that each subview is the same width as the NSSplitView, and the total of the subviews' heights is equal to the height of the NSSplitView. The NSSplitView positions its subviews so that the first subview is at the top of the NSSplitView, and each successive subview is positioned below the previous one. The user can set the height of two subviews by moving a horizontal bar called the *divider*, which makes one subview smaller and the other larger.

To add a view to an NSSplitView, you use the NSView method **addSubview:**. When the NSSplitView is displayed, it checks to see if its subviews are properly tiled. If not, it invokes the delegate method **splitView:resizeSubviewsWithOldSize:**, allowing the delegate to specify the heights of specific subviews. If the delegate doesn't implement this method, the NSSplitView sends **adjustSubviews** to itself to yield the default tiling behavior.

When a mouse-down occurs in an NSSplitView's divider, the NSSplitView determines the limits of the divider's travel and tracks the mouse to allow the user to drag the divider within these limits. With the following mouse-up, the NSSplitView resizes the two affected subviews, informs the delegate that the subviews were resized, and displays the affected views and divider. The NSSplitView's delegate can constrain the travel of specific dividers by implementing the method **splitView:constrainMinCoordinate:maxCoordinate:ofSubviewAt:**.

Managing Component Views

– (void)**adjustSubviews**	Adjusts the heights of the subviews.
– (float)**dividerThickness**	Returns the thickness of the divider.
– (void)**drawDividerInRect:**(NSRect)*aRect*	Draws the divider in *aRect*.

Assigning a Delegate

– (id)**delegate**	Returns the NSSplitView's delegate.
– (void)**setDelegate:**(id)*anObject*	Sets the NSSplitView's delegate.

Implemented by the Delegate

– (void)**splitView:**(NSSplitView *)*splitView* **constrainMinCoordinate:**(float *)*min* **maxCoordinate:**(float *)*max* **ofSubviewAt:**(int)*offset*

Sent directly by *splitView* to the delegate. Allows the delegate to constrain further *min* and *max* vertical travel of a divider. *offset* is an index that identifies the dividers in a NSSplitView from top to bottom starting with divider 0.

– (void)**splitView:**(NSSplitView *)*sender* **resizeSubviewsWithOldSize:**(NSSize)*oldSize*

Sent directly by *splitView* to the delegate. Allows the delegate to add custom resizing behavior after users resize an NSSplitView. *oldSize* is the size of the NSSplitView before the user resized it.

– (void)**splitViewDidResizeSubviews:**(NSNotification *)*notification*

Sent by the default notification center to the delegate; *aNotification* is always NSSplitViewDidResizeSubviewsNotification. If the delegate implements this method, it's automatically registered to receive this notification.

– (void)**splitViewWillResizeSubviews:**(NSNotification *)*notification*

Sent by the default notification center to the delegate; *aNotification* is always NSSplitViewWillResizeSubviewsNotification. If the delegate implements this method, it's automatically registered to receive this notification.

NSText

Inherits From: NSView : NSResponder : NSObject

Conforms To: NSChangeSpelling, NSIgnoreMisspelledWords
NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSTextView.h

Class Description

The NSText class declares the programmatic interface to objects that manage text. NSText objects are used by the Application Kit wherever text appears in interface objects: An NSText object draws the title of a window, the commands in a menu, the title of a button, and the items in a browser. Your application inherits these uses of the NSText class when it incorporates any of these objects into its interface. Your application can also create NSText objects for its own purposes.

The NSText class is unlike most other classes in the Application Kit in its complexity and range of features. One of its design goals is to provide a comprehensive set of text-handling features so that you'll rarely need to create a subclass. An NSText object can (among other things):

- Control the color of its text and background.
- Control the font and layout characteristics of its text.
- Control whether text is editable.
- Wrap text on a word or character basis.
- Display graphic images within its text.
- Write text to or read text from files in the form of RTFD—Rich Text Format files that contain TIFF or EPS images.
- Let another object, the delegate, dynamically control its properties.
- Let the user copy and paste text within and between applications.
- Let the user copy and paste font and format information between NSText objects.
- Let the user check the spelling of words in its text.
- Let the user control the format of paragraphs by manipulating a ruler.

Graphical user-interface building tools (such as Interface Builder) may give you access to NSText objects in several different configurations, such as those found in the NSTextField, NSForm, and NSScrollView objects. These classes configure an NSText object for their own specific purposes. Additionally, all NSTextFields, NSForms, NSButtons within the same window—in short, all objects that access an NSText object through associated Cells—share the same NSText object, reducing the memory demands of an application. Thus, it's generally best to use one of these classes whenever it meets your needs, rather than create NSText objects yourself. If one of these classes doesn't provide enough flexibility for your purposes, you can create NSText objects programatically.

Plain and Rich NSText Objects

When you create an NSText object directly, by default it allows only one font, line height, text color, and paragraph format for the entire text. Once an NSText object is created, you can alter its global settings using methods such as **setFont:** and **setTextColor:**. For convenience, such an NSText object will be called a *plain* NSText object.

To allow multiple values for attributes such as font and color, you must send the NSText object a **setRichText:YES** message. An NSText object that allows multiple fonts also allows multiple paragraph formats, line heights, and so on. For convenience, such an NSText object will be called a *rich* NSText object.

A rich NSText object can use RTF (Rich Text Format) as an interchange format. Not all RTF control words are supported: On input, an NSText object ignores any control word it doesn't recognize; some of those it can read and interpret it doesn't write out. These are the RTF control words that an NSText object recognizes.

Control Word	Read	Write
\ansi	yes	yes
\b	yes	yes
\cb	yes	yes
\cf	yes	yes
\colortbl	yes	yes
\dnn	yes	yes
\fin	yes	yes
\fn	yes	yes
\fonttbl	yes	yes
\fsn	yes	yes
\i	yes	yes
\lin	yes	yes
\margrn	yes	yes
\paperwn	yes	yes
\mac	yes	no
\margln	yes	yes
\par	yes	yes
\pard	yes	no
\pca	yes	no
\qc	yes	yes
\ql	yes	yes
\qr	yes	yes
\sn	yes	no
\tab	yes	yes
\upn	yes	yes

NSText objects are designed to work closely with various other objects. Some of these—such as the delegate or an embedded graphic object—require a degree of programming on your part. Others—such as the Font panel, spelling checker, or ruler—take no effort other than deciding whether the service should be enabled or disabled. The following sections discuss these interrelationships.

Notifying the NSText Object's Delegate

Many of an NSText object's actions can be controlled through an associated object, the NSText object's delegate. If it implements any of the following methods, the delegate receives the corresponding message at the appropriate time:

textDidBeginEditing:
textDidChange:
textDidEndEditing:
textShouldBeginEditing:
textShouldEndEditing:

So, for example, if the delegate implements the **textDidBeginEditing:** method, it will receive notification upon the user's first attempt to change the text. Moreover, depending on the method's return value, the delegate can either allow or prohibit changes to the text. See "Methods Implemented by the Delegate". The delegate can be any object you choose, and one delegate can control multiple NSText objects.

Adding Graphics to the Text

A rich NSText object allows graphics to be embedded in the text. Each graphic is treated as a single (possibly large) "character": The text's line height and character placement are adjusted to accommodate the graphic "character." Graphics are embedded in the text in either of two ways: programmatically or directly through user actions. In the programmatic approach, graphic objects are added using the **replaceRange:WithRTFD:** method.

An alternate means of adding an image to the text is for the user to drag an EPS or TIFF file icon directly into an NSText object. The NSText object automatically creates a graphic object to manage the display of the image. This feature requires a rich NSText object that has been configured to receive dragged images—see the **setImportsGraphics:** method.

Images that have been imported in this way can be written as RTFD documents. Programmatic creation of RTFD documents is not supported in this version of OpenStep. RTFD documents use a file package, or directory, to store the components of the document (the "D" stands for "directory"). The file package has the name of the document plus a ".rtfd" extension. The file package always contains a file called TXT.rtf for the text of the document, and one or more TIFF or EPS files for the images. An NSText object can transfer information in an RTFD document to a file and read it from a file—see the **writeRTFDToFile:atomically:** and **readRTFDFromFile:** methods.

Cooperating with Other Objects and Services

NSText objects are designed to work with the Application Kit's font conversion system. By default, an NSText object keeps the Font panel updated with the font of the current selection. It also changes the font of the selection (for a rich NSText object) or of the entire text (for a default NSText object) to reflect the user's choices in the Font panel or menu. To disconnect an NSText object from this service, send it a **setUsesFontPanel:NO** message.

If an NSText object is a subview of an NSScrollView, it can cooperate with the NSScrollView to display and update a ruler that displays formatting information. The NSScrollView retiles its subviews to make room for the ruler, and the NSText object updates the ruler with the format information of the paragraph containing the selection. The **toggleRuler:** method controls the display of this ruler. Users can modify paragraph formats by manipulating the components of the ruler.

Coordinates and sizes mentioned in the method descriptions below are in PostScript units—1/72 of an inch.

Getting and Setting Contents

– (void)**replaceRange:**(NSRange)*range* **withRTF:**(NSData *)*rtfData*	Replaces the characters within the specified *range* of text with the RTF data *rtfData*.
– (void)**replaceRange:**(NSRange)*range* **withRTFD:**(NSData *)*rtfdData*	Replaces the characters within the specified *range* of text with the RTFD data *rtfdData*.
– (NSData *)**RTFDFromRange:**(NSRange)*range*	Extracts the specified *range* of RTFD text from the NSText object and returns an data object initialized with that text.
– (NSData *)**RTFFromRange:**(NSRange)*range*	Extracts the specified *range* of RTF text from the NSText object and returns a data object initialized with that text. This data is formatted according to the RTF file format.
– (void)**setText:**(NSString *)*string*	Sets the contents of the NSText object to be *string*.
– (void)**setText:**(NSString *)*string* **range:**(NSRange)*range*	Replaces the characters in the specified *range* of text in the NSText object to be *string.*
– (NSString *)**text**	Returns the contents of the NSText object as a immutable string object.

Managing Global Characteristics

– (NSTextAlignment)**alignment**	Returns how text in the NSText object is aligned between the margins.
– (BOOL)**drawsBackground**	Returns whether the NSText object draws its own background.
– (BOOL)**importsGraphics**	Returns whether the NSText object can accept images.
– (BOOL)**isEditable**	Returns whether users can edit the NSText object.
– (BOOL)**isRichText**	Returns whether the text in the NSText object is RTF.
– (BOOL)**isSelectable**	Returns whether users can select text in the NSText object.
– (void)**setAlignment:**(NSTextAlignment)*mode*	Sets how the text in the NSText object is aligned between the margins.
– (void)**setDrawsBackground:**(BOOL)*flag*	Sets whether the NSText object draws its own background.
– (void)**setEditable:**(BOOL)*flag*	Sets whether users can edit text in the NSText object.
– (void)**setImportsGraphics:**(BOOL)*flag*	Sets whether the NSText object can accept images.

– (void)**setRichText:**(BOOL)*flag*	Sets whether the text in the NSText object allows for multiple values of attributes, such as color and font (i.e. RTF).
– (void)**setSelectable:**(BOOL)*flag*	Sets whether users can select text in the NSText object.

Managing Font and Color

– (NSColor *)**backgroundColor**	Returns the background color for the NSText object.
– (void)**changeFont:**(id)*sender*	Initiates a font-change session.
– (NSFont *)**font**	Returns the default NSFont object for the NSText object.
– (void)**setBackgroundColor:**(NSColor *)*color*	Sets the background color for the NSText object.
– (void)**setColor:**(NSColor *)*color* **ofRange:**(NSRange)*range*	Sets the color for the specified *range* of text in the NSText object to *color.*
– (void)**setFont:**(NSFont *)*obj*	Sets the default NSFont object for the NSText object.
– (void)**setFont:**(NSFont *)*font* **ofRange:**(NSRange)*range*	Sets the font for the specified *range* of text in the NSText object to *font*.
– (void)**setTextColor:**(NSColor *)*color*	Sets the textual color for the NSText object.
– (void)**setUsesFontPanel:**(BOOL)*flag*	Sets whether the NSText object uses the font panel.
– (NSColor *)**textColor**	Returns the textual color for the NSText object.
– (BOOL)**usesFontPanel**	Returns whether the NSText object uses the font panel

Managing the Selection

– (NSRange)**selectedRange**	Returns the range of the selected text in the NSText object.
– (void)**setSelectedRange:**(NSRange)*range*	Sets the *range* of selected text in the NSText object.

Sizing the Frame Rectangle

– (BOOL)**isHorizontallyResizable**	Returns whether the frame width can change.
– (BOOL)**isVerticallyResizable**	Returns whether the frame height can change.
– (NSSize)**maxSize**	Gets the maximum size of the NSTextView's frame.
– (NSSize)**minSize**	Gets the minimum size of the NSTextView's frame.
– (void)**setHorizontallyResizable:**(BOOL)*flag*	Sets whether the frame's width can change.

– (void)**setMaxSize:**(NSSize)*newMaxSize* — Sets the maximum size of the NSText object to *newMaxSize*.

– (void)**setMinSize:**(NSSize)*newMinSize* — Sets the minimum size of the NSText object to *newMinSize*.

– (void)**setVerticallyResizable:**(BOOL)*flag* — Sets whether the frame's height can change.

– (void)**sizeToFit** — Resizes the frame to fit just around the text.

Responding to Editing Commands

– (void)**alignCenter:**(id)*sender* — Centers the selected text between the margins.

– (void)**alignLeft:**(id)*sender* — Aligns selected text to the left margin.

– (void)**alignRight:**(id)*sender* — Aligns selected text the right margin.

– (void)**copy:**(id)*sender* — Copies the selected text to the pasteboard.

– (void)**copyFont:**(id)*sender* — Copies the selected text's font to the pasteboard.

– (void)**copyRuler:**(id)*sender* — Copies the selected text's ruler to the pasteboard.

– (void)**cut:**(id)*sender* — Deletes the selected text and copies it to the pasteboard.

– (void)**delete:**(id)*sender* — Deletes the selected text. This method posts the notification NSTextDidChangeNotification with the receiving object to the default notification center and may post the NSTextDidBeginEditing notification as well. (NSTextDidEndEditingNotification gets posted when the first responder changes.)

– (void)**paste:**(id)*sender* — Replaces the selected text with the contents of the pasteboard. This method posts the notification NSTextDidChangeNotification with the receiving object to the default notification center and may post the NSTextDidBeginEditing notification as well.

– (void)**pasteFont:**(id)*sender* — Replaces the selection's font with the pasteboard contents. This method posts the NSTextDidChangeNotification notification with the receiving object to the default notification center and may post the NSTextDidBeginEditing notification as well.

– (void)**pasteRuler:**(id)*sender* — Replaces the selection's ruler with the pasteboard contents.

– (void)**selectAll:**(id)*sender* — Selects all text in the NSText object.

– (void)**subscript:**(id)*sender* — Subscripts the current selection.

– (void)**superscript:**(id)*sender* — Superscripts the current selection.

– (void)**underline:**(id)*sender* — Underlines the selected text.

– (void)**unscript:**(id)*sender* — Removes superscript or subscript in the current selection.

Managing the Ruler

– (BOOL)**isRulerVisible** — Returns whether the ruler is visible.

– (void)**toggleRuler:**(id)*sender* — Displays the ruler if it's not visible, and removes it if it is visible.

Spelling

– (void)**checkSpelling:**(id)*sender* — Initiates a spell-checking session.

– (void)**showGuessPanel:**(id)*sender* — Displays the spell-checker's Show Guess panel.

Scrolling

– (void)**scrollRangeToVisible:**(NSRange)*range* — Scrolls the NSText object so that the *range* of text is visible.

Reading and Writing RTFD Files

– (BOOL)**readRTFDFromFile:**(NSString *)*path* — Reads RTFD data from the file package specified by *path* and initializes an NSText object with it; returns whether the operation succeeded.

– (BOOL)**writeRTFDToFile:**(NSString *)*path* **atomically:**(BOOL)*flag* — Writes RTFD data from the receiving NSText object to the file package specified by *path*. *flag* determines whether writing occurs atomically. Returns whether the operation succeeded.

Managing the Field Editor

– (BOOL)**isFieldEditor** — Returns whether the receiving NSText object gives up First Responder status on tab, carriage return, etc.

– (void)**setFieldEditor:**(BOOL)*flag* — Sets whether the receiving NSText object is to be used as a field editor. *flag* indicates whether to end on carriage return, tab, or other terminating character.

Managing the Delegate

– (id)**delegate** — Returns the delegate of the NSText object.

– (void)**setDelegate:**(id)*anObject* — Makes *anObject* the NSText object's delegate.

Implemented by the Delegate

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

Sent by the default notification center to the delegate; *aNotification* is always NSTextDidBeginEditingNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSTextDidChangeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to the delegate; *aNotification* is always NSTextDidEndEditingNotification. If the delegate implements this method, it's automatically registered to receive this notification.

– (BOOL)**textShouldBeginEditing:**(NSText *)*textObject*

Sent directly by *textObject* to the delegate. Informs delegate of an impending textual change. YES means go ahead and make the change.

– (BOOL)**textShouldEndEditing:**(NSText *)*textObject*

Sent directly by *textObject* to the delegate. Warns delegate of the impending loss of First Responder status. YES means go ahead and change status.

NSTextField

Inherits From: NSControl : NSView : NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSTextField.h

Class Description

An NSTextField is an NSControl object that can display a piece of text that a user can select or edit, and which sends an action message to its target if the user hits the Return key while editing. An NSTextField can also be linked to other NSTextFields, so that when the user presses Tab or Shift-Tab, the object assigned as the "next" or "previous" field gets a message to select its text.

An NSTextField is a good alternative to an NSText object for small regions of editable text, since the display of the NSTextField is achieved by using a global NSText object shared by objects all over your application, which saves on memory usage. Each NSWindow also has an NSText object used for editing of NSTextFields (and NSTextFieldCells in NSMatrices). An NSWindow's global NSText object is called a *field editor*, since it's attached as needed to an NSTextField to perform its editing. NSTextField allows you to specify an object to act as an indirect delegate to the field editor; the NSTextField itself acts as the NSText delegate if it needs to, then passes the delegate method on to its own NSText delegate.

Setting User Access to Text

– (BOOL)**isEditable**	Returns whether the NSTextField's text is editable.
– (BOOL)**isSelectable**	Returns whether the NSTextField's text is selectable.
– (void)**setEditable:**(BOOL)*flag*	Sets whether the NSTextField's text is editable.
– (void)**setSelectable:**(BOOL)*flag*	Sets whether the NSTextField's text is selectable.

Editing Text

– (void)**selectText:**(id)*sender*	Selects all of the text if it's selectable or editable.

Setting Tab Key Behavior

– (id)**nextText**	Gets the object selected when the user presses Tab.
– (id)**previousText**	Gets the object selected when the user presses Shift-Tab.
– (void)**setNextText:**(id)*anObject*	Sets the object selected when the user presses Tab.

– (void)**setPreviousText:**(id)*anObject* Sets the object selected when the user presses Shift-Tab.

Assigning a Delegate

– (void)**setDelegate:**(id)*anObject* Sets the delegate for messages from the field editor to *anObject*.

– (id)**delegate** Returns the delegate for messages from the field editor.

Modifying Graphic Attributes

– (NSColor *)**backgroundColor** Returns the color of the background.

– (BOOL)**drawsBackground** Returns whether the NSTextField draws its own background.

– (BOOL)**isBezeled** Returns whether the NSTextField has a bezeled border.

– (BOOL)**isBordered** Returns whether the NSTextField has a plain border.

– (void)**setBackgroundColor:**(NSColor *)*aColor* Sets the color of the background to *aColor.*

– (void)**setBezeled:**(BOOL)*flag* Sets whether the NSTextField has a bezeled border.

– (void)**setBordered:**(BOOL)*flag* Sets whether the NSTextField has a plain border.

– (void)**setDrawsBackground**:(BOOL)*flag* Sets whether the NSTextField draws its own background color.

– (void)**setTextColor:**(NSColor *)*aColor* Sets the color of the NSTextField's text to *aColor.*

– (NSColor *)**textColor** Returns the color of the NSTextField's text.

Target and Action

– (SEL)**errorAction** Returns the action method sent for an invalid value.

– (void)**setErrorAction:**(SEL)*aSelector* Sets the action method sent (*aSelector*) for an invalid value entered.

Handling Events

– (BOOL)**acceptsFirstResponder** Return YES if text is editable or selectable.

– (void)**textDidBeginEditing:**(NSNotification *)*notification*

Invoked when there’s a change in the text after the receiver gains first responder status. The default behavior is to pass this message on to the text delegate by posting the notification NSControlTextDidEndEditingNotification with the receiving object and, in the notification’s dictionary, the text object (with the key NSFieldEditor) to the default notification center.

– (void)**textDidChange:**(NSNotification *)*notification*

Invoked upon a key-down event or paste operation that changes the receiver’s contents. The default behavior is to pass this message on to the text delegate by posting the NSControlTextDidChangeNotification notification with the receiving object and, in the notification’s dictionary, the text object (with the key NSFieldEditor) to the default notification center.

– (void)**textDidEndEditing:**(NSNotification *)*notification*

Invoked when text editing ends. The default behavior is to pass this message on to the text delegate by posting the notification NSControlTextDidEndEditingNotification with the receiving object and, in the notification’s dictionary, the text object (with the key NSFieldEditor) to the default notification center.

– (BOOL)**textShouldBeginEditing:**(NSText *)*textObject*

Invoked to let the NSTextField respond to impending changes to its text and then forwarded to the text delegate.

– (BOOL)**textShouldEndEditing:**(NSText *)*textObject*

Invoked to let the NSTextField respond to impending loss of first responder status and then forwarded to the text delegate.

NSTextFieldCell

Inherits From: NSActionCell : NSCell : NSObject

Conforms To: NSCoding, NSCopying (NSCell)
NSObject (NSObject)

Declared In: AppKit/NSTextFieldCell.h

Class Description

NSCells display text or images—an NSTextFieldCell is simply an NSCell that displays text and that keeps track of its background and text colors. Normally, the NSCell class assumes white as the background when bezeled, and light gray otherwise, and the text is always black. With NSTextFieldCell, you can specify those colors.

Modifying Graphic Attributes

– (NSColor *)**backgroundColor**	Returns the color of the background.
– (BOOL)**drawsBackground**	Returns whether the NSTextFieldCell draws its own background.
– (void)**setBackgroundColor:**(NSColor *)*aColor*	Sets the color of the background to *aColor.*
– (void)**setDrawsBackground**:(BOOL)*flag*	Sets whether the NSTextFieldCell draws its own background.
– (void)**setTextColor:**(NSColor *)*aColor*	Sets the color of the text to *aColor.*
– (id)**setUpFieldEditorAttributes:**(id)*textObject*	Sets text attributes of the field editor to be the same as those of *textObject*. Used to set the attributes of text such as color and background color, for which there are no explicit methods.
– (NSColor *)**textColor**	Returns the color of the text.

NSView

Inherits From: NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSView.h
AppKit/NSClipView.h

Class Description

NSView is an abstract class that provides its subclasses with a structure for drawing and for handling events. Any application that needs to display, print, or receive events must use NSView objects.

To be displayed, a view must be placed in a window (represented by an NSWindow object). All the views within a window are arranged in a hierarchy, with each view having a single *superview* and zero or more *subviews*. Each view has its own area to draw in and its own coordinate system, expressed as a transformation of its superview's coordinate system. An NSView object can scale, translate, or rotate its coordinates, or flip the polarity of its y-axis.

An NSView keeps track of its size and location in two ways: as a frame rectangle (expressed in its superview's coordinate system) and as a bounds rectangle (expressed in its own coordinate system). Both are represented by NSRect structures.

Subclasses of NSView typically override **drawRect:** to implement an object's distinctive appearance. They also frequently override one or more of NSView's or NSResponder's event-handling methods, to react to the user's manipulations of the mouse and keyboard.

Initializing NSView Objects

– (id)**initWithFrame:**(NSRect)*frameRect*	Initializes a new NSView object to the location and dimensions of *frameRect*.

Managing the NSView Hierarchy

– (void)**addSubview:**(NSView *)*aView*	Makes *aView* a subview of the receiving view object.
– (void)**addSubview:**(NSView *)*aView* **positioned:**(NSWindowOrderingMode)*place* **relativeTo:**(NSView *)*otherView*	Makes *aView* a subview of the receiving view object. It is positioned relative to *otherView* according to *place*.

– (NSView *)**ancestorSharedWithView:**(NSView *)*aView*
Returns the ancestor view shared by *aView* and the receiver; **self** if *aView* is the receiving view or if the receiving view is the ancestor of *aView*; *aView* if it is the superview of the receiving view; or **nil** in any other case.

– (BOOL)**isDescendantOf:**(NSView *)*aView* Returns whether *aView* is an ancestor of the receiver.

– (NSView *)**opaqueAncestor** Returns the receiver's nearest opaque ancestor.

– (void)**removeFromSuperview** Removes the receiver from the view hierarchy.

– (void)**replaceSubview:**(NSView *)*oldView* **with:**(NSView *)*newView* Replaces *oldView* with *newView.*

– (void)**sortSubviewsUsingFunction:**(int (*)(id ,id ,void *))*compare* **context:**(void *)*context* Sorts the receiving view's subviews using the sorting function *compare* and the context *context*. The first two arguments of the function are the views to be compared.

– (NSArray *)**subviews** Returns a mutable array of the receiving view object's subviews.

– (NSView *)**superview** Returns the receiving view object's superview.

– (NSWindow *)**window** Returns the window in which the view is displayed.

– (void)**viewWillMoveToWindow:**(NSWindow *)*newWindow*
Notifies the view that it will move to a new window.

Modifying the Frame Rectangle

– (float)**frameRotation** Returns the angle of the frame rectangle's rotation.

– (NSRect)**frame** Gets the view's frame rectangle.

– (void)**rotateByAngle:**(float)*angle* Rotates the view's frame rectangle by *angl*e. This method posts the NSViewFocusChangedNotification notification with the receiving object to the default notification center.

– (void)**setFrame:**(NSRect)*frameRect* Assigns the view a new frame rectangle.

– (void)**setFrameOrigin**:(NSPoint)*newOrigin* Sets the origin of the view's frame to *newOrigin*. This method posts the NSViewFrameChangedNotification and NSViewFocusChangedNotification notifications with the receiving object to the default notification center.

– (void)**setFrameRotation:**(float)*angle* — Rotates the view's frame to *angle*. This method posts the NSViewFocusChangedNotification notification with the receiving object to the default notification center.

– (void)**setFrameSize:**(NSSize)*newSize* — Resizes the view's frame to *newSize.* This method posts the NSViewFrameChangedNotification and NSViewFocusChangedNotification notifications with the receiving object to the default notification center.

Modifying the Coordinate System

– (float)**boundsRotation** — Returns the rotation of the view's coordinate system.

– (NSRect)**bounds** — Gets the view's bounds rectangle.

– (BOOL)**isFlipped** — Returns whether the view is flipped.

– (BOOL)**isRotatedFromBase** — Returns whether the view is rotated.

– (BOOL)**isRotatedOrScaledFromBase** — Returns whether the view is rotated or scaled.

– (void)**scaleUnitSquareToSize:**(NSSize)*newSize* — Scales the NSView's coordinate system unit size to *newSize.* This method posts the notification NSViewFocusChangedNotification with the receiving object to the default notification center.

– (void)**setBounds:**(NSRect)*aRect* — Sets the NSView's bounds rectangle to *aRect*.

– (void)**setBoundsOrigin:**(NSPoint)*newOrigin* — Sets the NSView's drawing origin to *newOrigin*. This method posts the NSViewFocusChangedNotification notification with the receiving object to the default notification center.

– (void)**setBoundsRotation:**(float)*angle* — Rotates the NSView's coordinate system to *angle*. This method posts the NSViewFocusChangedNotification notification with the receiving object to the default notification center.

– (void)**setBoundsSize:**(NSSize)*newSize* — Resizes the NSView's coordinate system to *newSize.* This method posts the NSViewFocusChangedNotification notification with the receiving object to the default notification center.

– (void)**translateOriginToPoint:**(NSPoint)*point* — Shifts the NSView's coordinate system to *point*. This method posts the NSViewFocusChangedNotification notification with the receiving object to the default notification center.

Converting Coordinates

– (NSRect)**centerScanRect:**(NSRect)*aRect* — Converts the rectangle *aRect* to lie on centers of pixels.

– (NSPoint)**convertPoint:**(NSPoint)*aPoint* **fromView:**(NSView *)*aView* — Converts *aPoint* in *aView* to the receiver's coordinates.

– (NSPoint)**convertPoint:**(NSPoint)*aPoint* **toView:**(NSView *)*aView* — Converts *aPoint* in the receiver to *aView*'s coordinates.

– (NSRect)**convertRect:**(NSRect)*aRect* **fromView:**(NSView *)*aView* — Converts the rectangle *aRect* in *aView* to the receiver's coordinates.

– (NSRect)**convertRect:**(NSRect)*aRect* **toView:**(NSView *)*aView* — Converts the rectangle *aRect* in the receiver to *aView*'s coordinates.

– (NSSize)**convertSize:**(NSSize)*aSize* **fromView:**(NSView *)*aView* — Converts *aSize* in *aView* to the receiver's coordinates.

– (NSSize)**convertSize:**(NSSize)*aSize* **toView:**(NSView *)*aView* — Converts *aSize* in the receiver to *aView*'s coordinates.

Notifying Ancestor Views

– (BOOL)**postsFrameChangedNotifications** — Returns whether notifications of frame changes to ancestors are activated.

– (void)**setPostsFrameChangedNotifications:**(BOOL)*flag* — Sets whether to activate ancestor notifications.

Resizing Subviews

– (void)**resizeSubviewsWithOldSize:**(NSSize)*oldSize* — Initiates **superviewSizeChanged:** messages to subviews.

– (void)**setAutoresizesSubviews:**(BOOL)*flag* — Sets whether to notify subviews of resizing.

– (BOOL)**autoresizesSubviews** — Returns whether the NSView notifies subviews of resizing.

– (void)**setAutoresizingMask:**(unsigned int)*mask* — Determines automatic resizing behavior.

– (unsigned int)**autoresizingMask** — Returns the NSView's autosizing mask.

– (void)**resizeWithOldSuperviewSize:**(NSSize)*oldSize* — Notifies subviews that the superview changed size.

Graphics State Objects

– (void)**allocateGState**	Allocates a graphics state object.
– (void)**releaseGState**	Release the NSView's graphics state object.
– (int)**gState**	Returns the NSView's graphics state object.
– (void)**renewGState**	Marks the NSView's graphics state object as needing initialization.
– (void)**setUpGState**	Sets up the NSView's graphics state object.

Focusing

+ (NSView *)**focusView**	Returns the currently focused view.
– (void)**lockFocus**	Brings the receiving view into focus.
– (void)**unlockFocus**	Unfocuses the receiving view.

Displaying

– (BOOL)**canDraw**	Returns whether the view object can draw.
– (void)**display**	Displays the receiving view and its subviews.
– (void)**displayIfNeeded**	Conditionally displays the receiving view and its subviews (if opaque).
– (void)**displayIfNeededIgnoringOpacity**	Conditionally displays the receiving view and its subviews, regardless of opacity.
– (void)**displayRect:**(NSRect)*aRect*	Displays the receiving view and its subviews (if opaque) within *aRect*.
– (void)**displayRectIgnoringOpacity:**(NSRect)*aRect*	Displays the receiving view and its subviews (regardless of opacity) within *aRect*.
– (void)**drawRect:**(NSRect)*rect*	Implemented by subclasses to supply drawing instructions.
– (NSRect)**visibleRect**	Gets the receiving view's visible portion.
– (BOOL)**isOpaque**	Returns whether the view is opaque.
– (BOOL)**needsDisplay**	Returns whether the view needs to be redisplayed.
– (void)**setNeedsDisplay:**(BOOL)*flag*	If *flag* is YES, marks the view as changed, needing redisplay.

– (void)**setNeedsDisplayInRect:**(NSRect)*invalidRect* Marks the view as changed, needing redisplay in rectangle *invalidRect*.

– (BOOL)**shouldDrawColor** Returns whether the view should be drawn in color.

Scrolling

– (NSRect)**adjustScroll:**(NSRect)*newVisible* Lets the view object adjust the visible rectangle.

– (BOOL)**autoscroll:**(NSEvent *)*theEvent* Scrolls in response to a mouse-dragged event.

– (void)**reflectScrolledClipView**:(NSClipView *)*aClipView*
Reflects scrolling within clip view *aClipView.*

– (void)**scrollClipView:**(NSClipView *)*aClipView* **toPoint:**(NSPoint)*aPoint* Scrolls the clip view *aClipView* to *aPoint.*

– (void)**scrollPoint:**(NSPoint)*aPoint* Aligns *aPoint* with the content view's origin.

– (void)**scrollRect:**(NSRect)*aRect* **by:**(NSSize)*delta* Shifts the rectangle *aRect* by *delta*.

– (BOOL)**scrollRectToVisible:**(NSRect)*aRect* Scrolls the view so the rectangle *aRect* is visible.

Managing the Cursor

– (void)**addCursorRect:**(NSRect)*aRect* **cursor:**(NSCursor *)*anObject* Adds a cursor rectangle *aRect* for cursor *anObject* to the NSView.

– (void)**discardCursorRects** Removes all cursor rectangles in the view.

– (void)**removeCursorRect:**(NSRect)*aRect* **cursor:**(NSCursor *)*anObject* Removes cursor rectangle *aRect* for cursor *anObject* from the view.

– (void)**resetCursorRects** Implemented by subclasses to reset their cursor rectangles.

Assigning a Tag

– (int)**tag** Returns the view object's tag.

– (id)**viewWithTag:**(int)*aTag* Returns the subview object with *aTag* as its tag.

Aiding Event Handling

– (BOOL)**acceptsFirstMouse:**(NSEvent *)*theEvent* Returns whether the view object accepts first mouse-down events.

– (NSView *)**hitTest:**(NSPoint)*aPoint* Returns the lowest subview containing the point *aPoint*.

– (BOOL)**mouse:**(NSPoint)*aPoint* **inRect:**(NSRect)*aRect* Returns whether the point *aPoint* lies inside the *aRect*.

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

Implemented by subclasses to perform key-equivalent commands. Returns whether a subview handled *theEvent*.

- (void)**removeTrackingRect:**(NSTrackingRectTag)*tag*

Removes the tracking rectangle identified by *tag* from the view. (*tag* is an unique identifier returned from the **addTractingRect:owner:assumeInside:** method.)

– (BOOL)**shouldDelayWindowOrderingForEvent:**(NSEvent *)*anEvent*

Returns whether the view's window is brought forward normally (mouse-down) or delayed (mouse-up).

– (NSTrackingRectTag)**addTrackingRect:**(NSRect)*aRect*
owner:(id)*anObject*
userData:(void *)*data*
assumeInside:(BOOL)*flag*

Adds a tracking rectangle (*aRect*) owned by *anObject* to the receiving NSView. *flag* indicates whether the tracking rectangle will be only inside the NSView. Returns a unique tag that identifies the tracking rectangle.

Dragging

– (BOOL)**dragFile:**(NSString *)*filename*
fromRect:(NSRect)*rect*
slideBack:(BOOL)*slideFlag*
event:(NSEvent *)*event*

Initiates a file-dragging session, dragging file indicated by path *filename*. *rect* describes the position of the icon in the View's coordinates**.** *slideFlag* determines whether the NSImage should slide back if rejected

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

Initiates an image-dragging session, dragging *anImage* from *viewLocation*. *initialOffset* is the difference in the mouse location from the mouse-down. *pboard* is the pasteboard holding the data. *sourceObject* is the object receiving NSDraggingSource messages. *slideFlag* determines whether the NSImage should slide back if rejected.

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

Registers the pasteboard types that the window will accept in an image-dragging session.

– (void)**unregisterDraggedTypes**

Unregisters the window as a recipient of dragged images.

Printing

– (NSData *)**dataWithEPSInsideRect:**(NSRect)*aRect*	Returns a data object initialized with the EPS data within *aRect* in the receiving view.
– (void)**fax:**(id)*sender*	Faxes the view and its subviews.
– (void)**print:**(id)*sender*	Prints the view and its subviews.
– (void)**writeEPSInsideRect:**(NSRect)*rect* **toPasteboard:**(NSPasteboard *)*pasteboard*	Places PostScript code for the rectangle *rect* on the *pasteboard*.

Pagination

– (void)**adjustPageHeightNew:**(float *)*newBottom* **top:**(float)*oldTop* **bottom:**(float)*oldBottom* **limit:**(float)*bottomLimit*	Assists automatic pagination of the view object.
– (void)**adjustPageWidthNew:**(float *)*newRight* **left:**(float)*oldLeft* **right:**(float)*oldRight* **limit:**(float)*rightLimit*	Assists automatic pagination of the view object.
– (float)**heightAdjustLimit**	Returns how much of a page can go on the next page.
– (BOOL)**knowsPagesFirst:**(int *)*firstPageNum* **last:**(int *)*lastPageNum*	Returns whether the view paginates itself.
– (NSPoint)**locationOfPrintRect:**(NSRect)*aRect*	Locates the printing rectangle on the page.
– (NSRect)**rectForPage:**(int)*page*	Provides how much of the view will print on page.
– (float)**widthAdjustLimit**	Returns how much of a page can go on the next page.

Writing Conforming PostScript

– (void)**addToPageSetup**	Allows you to adjust for differences in the graphics state between the screen and the printer.
– (void)**beginPage:**(int)*ordinalNum* **label:**(NSString *)*aString* **bBox:**(NSRect)*pageRect* **fonts:**(NSString *)*fontNames*	Writes a page separator.
– (void)**beginPageSetupRect:**(NSRect)*aRect* **placement:**(NSPoint)*location*	Writes the beginning of a page setup section.

– (void)**beginPrologueBBox:**(NSRect)*boundingBox*
creationDate:(NSString *)*dateCreated*
createdBy:(NSString *)*anApplication*
fonts:(NSString *)*fontNames*
forWhom:(NSString *)*user*
pages:(int)*numPages*
title:(NSString *)*aTitle*

Writes the header for a print job.

– (void)**beginSetup**

Writes the beginning of the job setup section.

– (void)**beginTrailer**

Writes the beginning of the trailer for the print job.

– (void)**drawPageBorderWithSize:**(NSSize)*borderSize*

Implemented by subclasses to draw in margins (e.g., borders, numbering). *borderSize* is the size of the border.

– (void)**drawSheetBorderWithSize:**(NSSize)*borderSize*

Implemented by subclasses to draw in margins (e.g., borders, numbering). *borderSize* is the size of the border.

– (void)**endHeaderComments**

Writes the end of the header.

– (void)**endPrologue**

Writes the end of the prologue.

– (void)**endSetup**

Writes the end of the job setup section.

– (void)**endPageSetup**

Writes the end of a page setup section.

– (void)**endPage**

Writes the end of a page.

– (void)**endTrailer**

Writes the end of the trailer.

NSWindow

Inherits From: NSResponder : NSObject

Conforms To: NSCoding (NSResponder)
NSObject (NSObject)

Declared In: AppKit/NSWindow.h

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 window. The two principle functions of an NSWindow are to provide an area in which views can be placed, and to accept and distribute, to the appropriate NSViews, events that the user instigates by manipulating the mouse and keyboard.

Rectangles, Views, and the View Hierarchy

An NSWindow is defined by a *frame rectangle* that encloses the entire window, including its title bar, resize bar, and border, and by a *content rectangle* that encloses just its content area. Both rectangles are specified in the screen coordinate system. The frame rectangle establishes the NSWindow's *base coordinate system*. This coordinate system is always aligned with and is 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 a base coordinate system is the bottom left corner of the window's frame rectangle.

You create an NSWindow (through one of the **init:...** 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.

When it's created, an NSWindow automatically creates two NSViews: an opaque *frame view* and a transparent *content view* that fills the content area. The frame view is a private object that your application can't access directly. The content view is the "highest" accessible view in the window; you can replace the content view with an NSView of your own creation through NSWindow's **setContentView:** method.

You add other views to the window by declaring each to be a subview of the content view, or a subview of one of the content view's subviews, and so on, through NSView's **addSubview:** method. This tree of views is called the window's *view hierarchy.* When an NSWindow is told to display itself, it does so by sending view-displaying messages to each object 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).

Event Handling

The window system and the NSApplication object forward mouse and keyboard events to the appropriate NSWindow object. The NSWindow that's currently designated to receive keyboard events is known as the *key window.* If the mouse or keyboard event affects the window directly—resizing or moving it, for example—the NSWindow performs the appropriate operation itself and sends messages to its delegate informing it of its intentions, thus allowing your application to intercede. Events that are directed at specific views within the window are forwarded by the NSWindow to the NSView.

The NSWindow keeps track of the object that was last selected to handle keyboard events as its *first responder.* The first responder is typically the NSView that displays the current selection. In addition to keyboard events, the first responder is sent action messages that have a user-selected target (a **nil** target in program code). The NSWindow continually updates the first responder in response to the user's mouse actions.

Each NSWindow provides a *field editor,* an NSText object that handles small-scale text-editing chores. The field editor can be used by the NSWindow's first responder to edit the text that it displays. The **fieldEditor:forObject:** method returns the NSWindow's field editor. (You can make this method instead return an alternative NSText object, appropriate for the object specified the second argument, by implementing the delegate method **windowWillReturnFieldEditor:toObject:**.)

Initializing and Getting a New NSWindow Object

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

Initializes the new window object with a location and size for content of *contentRect*, a window style and buttons as indicated in the bitmap mask *aStyle*, drawing buffering as indicated by *bufferingType*. If *flag* is YES, the window system defers creating the window until it's needed.

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

Initializes the new window object for a screen as specified by *aScreen*, with a location and size for content of *contentRect*, a window style and buttons as indicated in the bitmap mask *aStyle*, drawing buffering as indicated by *bufferingType*. If *flag* is YES,the window system defers creating the window until it's needed.

Computing Frame and Content Rectangles

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

Gets the content rectangle for frame rectangle *aRect* in a window of type *aStyle*.

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

Gets the frame rectangle for content rectangle *aRect* in a window of type *aStyle*.

+ (float)**minFrameWidthWithTitle:**(NSString *)*aTitle*
styleMask:(unsigned int)*aStyle*
Returns the minimum frame width needed for *aTitle* in a window of type *aStyle*.

Accessing the Content View

– (id)**contentView**
Returns the NSWindow's content view.

– (void)**setContentView:**(NSView *)*aView*
Makes *aView* the NSWindow's content view.

Window Graphics

– (NSColor *)**backgroundColor**
Returns the window's background color.

– (NSString *)**representedFilename**
Returns the filename associated with this window (regardless of the title string).

– (void)**setBackgroundColor:**(NSColor *)*color*
Sets the window's background color to *color.*

– (void)**setRepresentedFilename:**(NSString *)*aString*
Alters *aString* by formatting it as a path and filename, then sets the filename associated with this window to the result. If *filename* doesn't include a path to the file, the current working directory is used. This method doesn't affect the title string.

– (void)**setTitle:**(NSString *)*aString*
Makes *aString* the window's title.

– (void)**setTitleWithRepresentedFilename:**(NSString *)*aString*
Invokes **setRepresentedFilename:** and makes the resultant string the window's title.

– (unsigned int)**styleMask**
Returns the window's border and title-bar style.

– (NSString *)**title**
Returns the window's title string.

Window Device Attributes

– (NSBackingStoreType)**backingType**
Returns the type of the window device's backing store.

– (NSDictionary *)**deviceDescription**
Returns the window device's attributes as key/value pairs.

– (int)**gState**
Returns the graphics-state object for the window object.

– (BOOL)**isOneShot**
Returns whether backing-store memory for the window is freed when the window is ordered off-screen.

– (void)**setBackingType:**(NSBackingStoreType)*type*
Sets the type of window-device backing store.

– (void)**setOneShot:**(BOOL)*flag*	Sets whether backing-store memory for the window should be freed when the window is ordered off-screen.
– (int)**windowNumber**	Returns the window number.

The Miniwindow

– (NSImage *)**miniwindowImage**	Returns the image that's displayed in the miniwindow.
– (NSString *)**miniwindowTitle**	Returns the title that's displayed in the miniwindow.
– (void)**setMiniwindowImage:**(NSImage *)*image*	Sets the *image* that's displayed in the miniwindow.
– (void)**setMiniwindowTitle:**(NSString *)*title*	Sets the *title* that's displayed in the miniwindow.

The Field Editor

– (void)**endEditingFor:**(id)*anObject*	Ends the field editor's editing assignment for *anObject*.
– (NSText *)**fieldEditor:**(BOOL)*createFlag* **forObject:**(id)*anObject*	Returns the window object's field editor for *anObject*. If the field editor does not exist and *createFlag* is YES, creates a field editor.

Window Status and Ordering

– (void)**becomeKeyWindow**	Records the window's new status as the key window. This method posts the notification NSWindowDidBecomeKeyNotification with the receiving object to the default notification center.
– (void)**becomeMainWindow**	Records the window's new status as the main window. This method posts the notification NSWindowDidBecomeMainNotification with the receiving object to the default notification center.
– (BOOL)**canBecomeKeyWindow**	Returns whether the receiving window object can be the key window.
– (BOOL)**canBecomeMainWindow**	Returns whether the receiving window object can be the main window.
– (BOOL)**hidesOnDeactivate**	Returns whether deactivation hides the window.
– (BOOL)**isKeyWindow**	Returns whether the receiving window object is the key window.
– (BOOL)**isMainWindow**	Returns whether the receiving window object is the main window.

– (BOOL)**isMiniaturized**	Returns whether the window is hidden (and the miniwindow displayed).
– (BOOL)**isVisible**	Returns whether the window object is in the screen list (and thus visible).
– (int)**level**	Returns the current window level.
– (void)**makeKeyAndOrderFront:**(id)*sender*	Makes the receiving window object the key window and brings it forward.
– (void)**makeKeyWindow**	Makes the receiving window object the key window.
– (void)**makeMainWindow**	Makes the receiving window object the main window.
– (void)**orderBack:**(id)*sender*	Puts the window object at the back of its tier.
– (void)**orderFront:**(id)*sender*	Puts the window object at the front of its tier.
– (void)**orderFrontRegardless**	Puts the window object at the front even if the application is inactive. If the window is currently miniaturized, this method posts the notification NSWindowDidDeminiaturizeNotification with the window object to the default notification center.
– (void)**orderOut:**(id)*sender*	Removes the window object from the screen list.
– (void)**orderWindow:**(NSWindowOrderingMode)*place* **relativeTo:**(int)*otherWin*	Repositions the window object in the screen list in position *place* relative to another window. If the window is currently miniaturized, this method posts the NSWindowDidDeminiaturizeNotification notification with that window object to the default notification center.
– (void)**resignKeyWindow**	Records that the window object is no longer the key window. This method posts the notification NSWindowDidResignKeyNotification with the receiving object to the default notification center.
– (void)**resignMainWindow**	Records that the window object is no longer the main window. This method posts the notification NSWindowDidResignMainNotification with the receiving object to the default notification center.
– (void)**setHidesOnDeactivate:**(BOOL)*flag*	Sets whether deactivation hides the window.
– (void)**setLevel**:(int)*newLevel*	Resets the window level to *newLevel*.

Moving and Resizing the Window

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

When successively invoked, tiles windows by offsetting them slightly to the right and down from the previous window. Returns the top left point of the placed window, which is typically used for *topLeftPoint* in the next invocation. If you specify (0,0), places the window as is, and returns its top left point.

– (void)**center**

Centers the window on the screen.

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

Constrains the window's frame rectangle *frameRect* to *screen*. Returns the frame rectangle.

– (NSRect)**frame**

Returns the window's frame rectangle

– (NSSize)**minSize**

Returns the window's minimum size.

– (NSSize)**maxSize**

Returns the window's maximum size

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

Resizes the window's content area to *aSize.*

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

Moves and/or resizes the window frame to *frameRect*. *flag* determines whether the window is displayed. This method posts the NSWindowDidResizeNotification notification with the receiving object to the default notification center.

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

Moves the window by changing its frame origin to *aPoint*.

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

Moves the window by changing its top-left corner to *aPoint*.

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

Sets the window's minimum size.

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

Sets the window's maximum size.

Converting Coordinates

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

Converts *aPoint* from base to screen coordinates.

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

Converts *aPoint* from screen to base coordinates.

Managing the Display

– (void)**display**	Displays all the window’s views.
– (void)**disableFlushWindow**	Disables flushing for a buffered window.
– (void)**displayIfNeeded**	Displays all the window’s views that need to be redrawn.
– (void)**enableFlushWindow**	Enables flushing for a buffered window.
– (void)**flushWindow**	Flushes the window’s buffer to the screen.
– (void)**flushWindowIfNeeded**	Conditionally flushes the window’s buffer to the screen.
– (BOOL)**isAutodisplay**	Returns whether the window displays all views requiring redrawing when **update** is invoked.
– (BOOL)**isFlushWindowDisabled**	Returns whether flushing is disabled.
– (void)**setAutodisplay**:(BOOL)*flag*	Sets whether the window displays all views requiring redrawing when **update** is invoked.
– (void)**setViewsNeedDisplay:**(BOOL)*flag*	Sets whether some views of the receiving window object should be redrawn.
– (void)**update**	Update’s the window’s display and cursor rectangles. This method is invoked after every event. When it successfully completes, it posts the NSWindowDidUpdateNotification notification.
– (void)**useOptimizedDrawing:**(BOOL)*flag*	Sets whether the window’s views should optimize drawing.
– (BOOL)**viewsNeedDisplay**	Returns whether some views of the receiving NSWindow object should be redrawn.

Screens and Window Depths

+ (NSWindowDepth)**defaultDepthLimit**	Returns the default depth limit for all windows.
– (BOOL)**canStoreColor**	Returns whether the window is deep enough to store colors.
– (NSScreen *)**deepestScreen**	Returns the deepest screen that the window is on.
– (NSWindowDepth)**depthLimit**	Returns the window’s depth limit.
– (BOOL)**hasDynamicDepthLimit**	Returns whether the depth limit depends on the screen.
– (NSScreen *)**screen**	Returns the screen that (most of) the window is on.
– (void)**setDepthLimit:**(NSWindowDepth)*limit*	Sets the window’s depth limit to *limit*
– (void)**setDynamicDepthLimit:**(BOOL)*flag*	Sets whether the depth limit will depend on the screen.

Cursor Management

– (BOOL)**areCursorRectsEnabled**	Returns whether cursor rectangles are enabled.
– (void)**disableCursorRects**	Disables all cursor rectangles in the window object.
– (void)**discardCursorRects**	Removes all cursor rectangles in the window object.
– (void)**enableCursorRects**	Enables cursor rectangles in the window object.
– (void)**invalidateCursorRectsForView:**(NSView *)*aView*	Marks cursor rectangles invalid for *aView*.
– (void)**resetCursorRects**	Resets cursor rectangles for the window object.

Handling User Actions and Events

– (void)**close**	Closes the window. When this method begins, it posts the notification NSWindowWillCloseNotification with the receiving object to the default notification center.
– (void)**deminiaturize:**(id)*sender*	Hides the miniwindow and redisplays the window.
– (BOOL)**isDocumentEdited**	Returns whether the window's document has been edited.
– (BOOL)**isReleasedWhenClosed**	Returns whether the window object is released when it is closed.
– (void)**miniaturize:**(id)*sender*	Hides the window and displays its miniwindow. When this method begins, it posts the notification NSWindowWillMiniaturizeNotification with the receiving object to the default notification center. When it completes successfully, it posts NSWindowDidMiniaturizeNotification.
– (void)**performClose:**(id)*sender*	Simulates user clicking the close button.
– (void)**performMiniaturize:**(id)*sender*	Simulates user clicking the miniaturize button.
– (int)**resizeFlags**	Returns the event modifier flags during resizing.
– (void)**setDocumentEdited:**(BOOL)*flag*	Sets whether the window's document has been edited.
– (void)**setReleasedWhenClosed:**(BOOL)*flag*	Sets whether closing the window object also releases it.

Aiding Event Handling

– (BOOL)**acceptsMouseMovedEvents**	Returns whether the NSWindow accepts mouse-moved events.
– (NSEvent *)**currentEvent**	Returns the current event object for the application.

– (void)**discardEventsMatchingMask:**(unsigned int)*mask*
beforeEvent:(NSEvent *)*lastEvent* — Discards any events in the event queue that have a type indicated by bitmap *mask* until the method encounters the event *lastEvent*.

– (NSResponder *)**firstResponder** — Returns the first responder to user events.

– (void)**keyDown:**(NSEvent *)*theEvent* — Handles key-down events.

– (BOOL)**makeFirstResponder:**(NSResponder *)*aResponder* — Makes *aResponder* the first responder to user events.

– (NSPoint)**mouseLocationOutsideOfEventStream** — Provides current location of the cursor.

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask* — Returns the next event object for the application that matches the events indicated by event mask *mask*.

– (NSEvent *)**nextEventMatchingMask:**(unsigned int)*mask*
untilDate:(NSDate *)*expiration*
inMode:(NSString *)*mode*
dequeue:(BOOL)*deqFlag* — Returns the next event object for the application that matches the events indicated by event mask *mask*, and that occurs before time *expiration*; until *expiration*, the run loop runs in *mode*.

– (void)**postEvent:**(NSEvent *)*event*
atStart:(BOOL)*flag* — Post an *event* for the application; if *atStart* is YES, the event goes to the beginning of the event queue.

– (void)**setAcceptsMouseMovedEvents:**(BOOL)*flag* — Sets whether the NSWindow accepts mouse-moved events.

– (void)**sendEvent:**(NSEvent *)*theEvent* — Dispatches mouse and keyboard events. If this method is dispatching a window exposed event, it posts the NSWindowDidExposeNotification notification with the receiving object and, in the notification's dictionary, a rectangle describing the exposed area (with the key NSExposedRect) to the default notification center. If it is dispatching a screen changed event, it posts NSWindowDidChangeScreenNotification with the receiving object. If it is dispatching a window moved event, it posts NSWindowDidMoveNotification.

– (BOOL)**tryToPerform:**(SEL)*anAction*
with:(id)*anObject* — Aids in dispatching action messages (*anAction*) to *anObject*.

– (BOOL)**worksWhenModal** — Override to return whether the window object accepts events when a modal panel is being run. Default is NO.

Dragging

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

Initiates an image-dragging session. NSView invokes this method inside its implementation of **mouseDown:**.

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

Registers the NSPasteboard types (*newTypes*) that the window object accepts in an image-dragging session.

– (void)**unregisterDraggedTypes**

Unregisters the window object as a recipient of dragged images.

Services and Windows Menu Support

– (BOOL)**isExcludedFromWindowsMenu**

Returns whether the receiving window object is omitted from the Windows menu.

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

Sets whether the receiving window object is omitted from the Windows menu.

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

Returns whether the window can respond to a service with send and receive types *sendType* and *returnType*.

Saving and Restoring the Frame

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

Removes the named frame rectangle from the system defaults.

– (NSString *)**frameAutosaveName**

Returns the name that's used to autosave the frame rectangle as a system default.

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

Saves the frame rectangle as a system default.

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

Sets the *name* that's used to autosave the frame rectangle as a system default.

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

Sets the frame rectangle from *string*, which encodes the position and dimensions of the frame rectangle and the position and dimensions of the screen.

– (BOOL)**setFrameUsingName:**(NSString *)*name* Sets the frame rectangle from the named default.

– (NSString *)**stringWithSavedFrame** Returns a string encoding the position and dimensions of the frame rectangle and the position and dimensions of the screen.

Printing and PostScript

– (NSData *)**dataWithEPSInsideRect:**(NSRect)*rect* Returns the encapsulated PostScript inside *rect* as a data object.

– (void)**fax:**(id)*sender* Faxes all the window's views.

– (void)**print:**(id)*sender* Prints all the window's views.

Assigning a Delegate

– (id)**delegate** Returns the window object's delegate.

– (void)**setDelegate:**(id)*anObject* Makes *anObject* the window object's delegate.

Implemented by the Delegate

– (BOOL)**windowShouldClose:**(id)*sender* Notifies delegate that the window is about to close.

– (NSSize)**windowWillResize:**(NSWindow *)*sender*
toSize:(NSSize)*frameSize* Lets delegate constrain resizing to *frameSize*.

– (id)**windowWillReturnFieldEditor:**(NSWindow *)*sender*
toObject:(id)*client* Lets delegate provide another text object for field editor.

– (void)**windowDidBecomeKey:**(NSNotification *)*aNotification*
Sent by the default notification center to notify the delegate that the window is the key window. *aNotification* is always NSWindowDidBecomeKeyNotification. If the delegate implements this method, it's automatically registered to receive this notification.

– (void)**windowDidBecomeMain:**(NSNotification *)*aNotification*
Sent by the default notification center to notify the delegate that the window is the main window. *aNotification* is always NSWindowDidBecomeMainNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window changed screens. *aNotification* is always NSWindowDidChangeScreenNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window was restored to screen. *aNotification* is always NSWindowDidDeminiaturizeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window was exposed. *aNotification* is always NSWindowDidExposeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window was miniaturized. *aNotification* is always NSWindowDidMiniaturizeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window did move. *aNotification* is always NSWindowDidMoveNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window isn't the key window. *aNotification* is always NSWindowDidResignKeyNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window isn't the main window. *aNotification* is always NSWindowDidResignMainNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window was resized. *aNotification* is always NSWindowDidResizeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window was updated. *aNotification* is always NSWindowDidUpdateNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window will close. *aNotification* is always NSWindowWillCloseNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window will be miniaturized. *aNotification* is always NSWindowWillMiniaturizeNotification. If the delegate implements this method, it's automatically registered to receive this notification.

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

Sent by the default notification center to notify the delegate that the window will move. *aNotification* is always NSWindowWillMoveNotification. If the delegate implements this method, it's automatically registered to receive this notification.

NSWorkspace

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: AppKit/NSWorkspace.h

Class Description

An NSWorkspace object responds to application requests to perform a variety of services:

- Opening, manipulating, and obtaining information about files and devices
- Tracking changes to the file system, devices, and the user database
- Launching applications
- Miscellaneous services such as animating an image and requesting additional time before power off

An NSWorkspace object is made available through the **sharedWorkspace** method. For example, the following statement uses an NSWorkspace object to request that a file be opened in the Edit application:

```
[[NSWorkspace sharedWorkspace] openFile:@"/Myfiles/README" withApplication:@"Edit"];
```

Creating a Workspace

+ (NSWorkspace *)**sharedWorkspace**	Returns a shared workspace.

Opening Files

– (BOOL)**openFile:**(NSString *)*fullPath*	Instructs Workspace Manager to open the file specified by *fullPath* using the default application for its type; returns YES if file was successfully opened and NO otherwise.
– (BOOL)**openFile:**(NSString *)*fullPath* **fromImage:**(NSImage *)*anImage* **at:**(NSPoint)*point* **inView:**(NSView *)*aView*	Instructs Workspace Manager to open the file specified by *fullPath* using the default application for its type. To provide animation prior to the open, *anImage* should contain the file's icon, and its image should be displayed at *point*, using *aView*'s coordinates. Returns YES if file was successfully opened and NO otherwise.

– (BOOL)**openFile:**(NSString *)*fullPath*
withApplication:(NSString *)*appName*

Instructs Workspace Manager to open the file specified by *fullPath* using the *appName* application; returns YES if file was successfully opened and NO otherwise.

– (BOOL)**openFile:**(NSString *)*fullPath*
withApplication:(NSString *)*appName*
andDeactivate:(BOOL)*flag*

Instructs Workspace Manager to open the file specified by *fullPath* using the *appName* application where *flag* indicates if sending application should be deactivated before the request is sent; returns YES if file was successfully opened and NO otherwise.

– (BOOL)**openTempFile:**(NSString *)*fullPath*

Instructs Workspace Manager to open the temporary file specified by *fullPath* using the default application for its type; returns YES if file was successfully opened and NO otherwise.

Manipulating Files

– (BOOL)**performFileOperation:**(NSString *)*operation*
source:(NSString *)*source*
destination:(NSString *)*destination*
files:(NSArray *)*files*
tag:(int *)*tag*

Requests the Workspace Manager to perform a file *operation* on a set of *files* in the *source* directory specifying the *destination* directory if needed using *tag* as an identifier for asynchronous operations; returns YES if operation succeeded and NO otherwise.

– (BOOL)**selectFile:**(NSString *)*fullPath*
inFileViewerRootedAtPath:(NSString *)*rootFullpath*

Instructs Workspace Manager to select the file specified by *fullPath* opening a new file viewer if a path is specified by *rootFullpath*; returns YES if file was successfully selected and NO otherwise.

Requesting Information about Files

– (NSString *)**fullPathForApplication:**(NSString *)*appName*

Returns the full path for the application *appName*.

– (BOOL)**getFileSystemInfoForPath:**(NSString *)*fullPath*
isRemovable:(BOOL *)*removableFlag*
isWritable:(BOOL *)*writableFlag*
isUnmountable:(BOOL *)*unmountableFlag*
description:(NSString **)*description*
type:(NSString **)*fileSystemType*

Describes the file system at *fullPath* in *description* and *fileSystemType*, sets the *Flags* appropriately, and returns YES if *fullPath* is a file system mount point, or NO if it isn't.

– (BOOL)**getInfoForFile:**(NSString *)*fullPath* **application:**(NSString **)*appName* **type:**(NSString **)*type*

Retrieves information about the file specified by *fullPath*, sets *appName* to the application the Workspace Manager would use to open *fullPath*, sets *type* to a value or file name extension indicating the file's type, and returns YES upon success and NO otherwise.

– (NSImage *)**iconForFile:**(NSString *)*fullPath*

Returns an NSImage with the icon for the single file specified by *fullPath*.

- (NSImage *)**iconForFiles:**(NSArray *)*pathArray*

Returns an NSImage with the icon for the files specified in *pathArray*, an array of NSStrings. If *pathArray* specifies one file, its icon is returned. If *pathArray* specifies more than one file, an icon representing the multiple selection is returned.

– (NSImage *)**iconForFileType:**(NSString *)*fileType*

Returns an NSImage the icon for the file type specified by *fileType*.

Tracking Changes to the File System

– (BOOL)**fileSystemChanged**

Returns whether a change to the file system has been registered with a **noteFileSystemChanged** message since the last **fileSystemChanged** message.

– (void)**noteFileSystemChanged**

Informs Workspace Manager that the file system has changed.

Updating Registered Services and File Types

– (void)**findApplications**

Instructs Workspace Manager to examine all applications in the normal places and update its records of registered services and file types.

Launching and Manipulating Applications

– (void)**hideOtherApplications**

Hides all applications other than the sender.

– (BOOL)**launchApplication:**(NSString *)*appName*

Instructs Workspace Manager to launch the application *appName* and returns YES if application was successfully launched and NO otherwise.

– (BOOL)**launchApplication:**(NSString *)*appName* **showIcon:**(BOOL)*showIcon* **autolaunch:**(BOOL)*autolaunch*

Instructs Workspace Manager to launch the application *appName* displaying the application's icon if *showIcon* is YES and using the dock autolaunching defaults if *autolaunch* is YES; returns YES if application was successfully launched and NO otherwise.

Unmounting a Device

– (BOOL)**unmountAndEjectDeviceAtPath:**(NSString *)*path*

Unmounts and ejects the device at *path* and returns YES if unmount succeeded and NO otherwise.

Tracking Status Changes for Devices

– (void)**checkForRemovableMedia**

Causes the Workspace Manager to poll the system's drives for any disks that have been inserted but not yet mounted. Asks the Workspace Manager to mount the disk asynchronously and returns immediately.

– (NSArray *)**mountNewRemovableMedia**

Causes the Workspace Manager to poll the system's drives for any disks that have been inserted but not yet mounted, waits until the new disks have been mounted, and returns a list of full pathnames to all newly mounted disks.

– (NSArray *)**mountedRemovableMedia**

Returns a list of the pathnames of all currently mounted removable disks.

Notification Center

– (NSNotificationCenter *)**notificationCenter**

Returns the notification center for WorkSpace notifications.

Tracking Changes to the User Defaults Database

– (void)**noteUserDefaultsChanged**

Informs Workspace Manager that the defaults database has changed.

– (BOOL)**userDefaultsChanged**

Returns whether a change to the defaults database has been registered with a **noteUserDefaultsChanged** message since the last **userDefaultsChanged** message.

Animating an Image

– (void)**slideImage:**(NSImage *)*image*
from:(NSPoint)*fromPoint*
to:(NSPoint)*toPoint*

Instructs Workspace Manager to animate a sliding image of *image* from *fromPoint* to *toPoint*, specified in screen coordinates.

Requesting Additional Time before Power Off or Logout

– (int)**extendPowerOffBy:**(int)*requested* Requests more time before the power goes off or the user logs out; returns the granted number of additional milliseconds.

Protocols

NSChangeSpelling

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

An object in the responder chain that can correct a misspelled word implements this protocol. See the description of the NSSpellChecker class for more information.

Changing Spelling

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

Implement to replace the selected word in the receiver with a corrected version from the Spelling panel. This message is sent by the NSSpellChecker instance to the object whose text is being checked. To get the corrected spelling, the receiver asks the sender for the string value of its selected cell.

NSColorPickingCustom

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

Together with the NSColorPickingDefault protocol, NSColorPickingCustom provides a way to add color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPicker class adopts the NSColorPickingDefault protocol. The easiest way to implement a color picker is to create a subclass of NSColorPicker and use it as a base upon which to add the NSColorPickingCustom protocol.

See also: NSColorPickingDefault, NSColorPicker (class)

Getting the Mode

– (int)**currentMode**

Returns the color picker's current mode (or submode, if applicable). The returned value should be unique to your color picker. (**NSColorPanel.h**. defines unique values for the standard color pickers used by the Application Kit.)

– (BOOL)**supportsMode:**(int)*mode*

Returns YES if the receiver supports the specified picking mode.

Getting the V iew

– (NSView *)**provideNewView:**(BOOL)*firstRequest*

Returns the view containing the color picker's user interface. This message is sent to the color picker whenever the color panel attempts to display it; the argument indicates whether this is the first time the message has been sent. If *firstRequest* is YES, the method should perform any initialization required (such as lazily loading a nib file).

Setting the Current Color

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

Adjusts the color picker to make *aColor* the currently selected color.

NSColorPickingDefault

Adopted By: NSColorPicker

Declared In: AppKit/NSColorPicking.h

Protocol Description

The NSColorPickingDefault protocol, together with the NSColorPickingCustom protocol, provides an interface for adding color pickers—custom user interfaces for color selection—to an application's NSColorPanel. The NSColorPickingDefault protocol provides basic behavior for a color picker. The NSColorPickingCustom protocol provides implementation-specific behavior.

The NSColorPicker class implements the NSColorPickingDefault protocol. The simplest way to implement your own color picker is to create a subclass of NSColorPicker, implementing the NSColorPickingCustom protocol in that subclass. However, it's possible to create a subclass of another class, such as NSView, and use it as a base upon which to add the methods of both NSColorPickingDefault and NSColorPickingCustom.

Color Picker Bundles

A class that implements the NSColorPickingDefault and NSColorPickingCustom protocols needs to be compiled and linked in an application's object file. However, your application need not explicitly create an instance of this class. Instead, your application's file package should include a directory named **ColorPickers**; within this directory you should place a directory *MyPickerClass***.bundle** for each custom color picker your application implements. This bundle should contain all resources required for your color picker: nib files, TIFF files, and so on.

NSColorPanel will allocate and initialize an instance of each class for which a bundle is found in the **ColorPickers** directory. The class name is assumed to be the bundle directory name minus the **.bundle** extension.

Color Picker Buttons

NSColorPanel lets the user select a color picker from a matrix of NSButtonCells. This protocol includes methods for providing and manipulating the image that gets displayed on the button.

See also: NSColorPickingCustom, NSColorPicker (class), NSColorPanel (class)

Initializing a Color Picker

– (id)**initWithPickerMask:**(int)*mask*
colorPanel:(NSColorPanel *)*colorPanel*

Initializes the receiver for the specified mask and color panel. This method is sent by the NSColorPanel to all implementors of the color picking protocols when the application's color panel is first initialized. If the color picker responds to any of the modes represented in *mask*, it should perform its initialization (if desired) and return **self**; otherwise it should do nothing and return **nil**. However, a custom color picker can instead delay initialization until it receives a **provideNewView:** message.

Adding Button Images

– (void)**insertNewButtonImage:**(NSImage *)*newImage*
in:(NSButtonCell *)*newButtonCell*

Sets *newImage* as *newButtonCell*'s image. *newButtonCell* is the NSButtonCell object that lets the user choose the picker from the color panel. This method should perform application-specific manipulation of the image before it's inserted and displayed by the button cell.

– (NSImage *)**provideNewButtonImage**

Returns the image for the mode button that the user uses to select this picker in the color panel. (This is the same image that the color panel uses as an argument when sending the **insertNewButtonImage:in:** message.)

Setting the Mode

– (void)**setMode:**(int)*mode*

Sets the color picker's mode. This method is invoked by NSColorPanel's **setMode:** method to ensure that the color picker reflects the current mode. Most color pickers have only one mode, and thus don't need to do any work in this method. Others, like the standard sliders picker, have multiple modes.

Using Color Lists

– (void)**attachColorList:**(NSColorList *)*aColorList*

Attaches the given color list to the receiver, if it isn't already displaying the list. This method is invoked automatically by the NSColorPanel when its **attachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself.

– (void)**detachColorList:**(NSColorList *)*aColorList* Removes the given color list from the receiver, unless the receiver isn’t displaying the list. This method is invoked automatically by the NSColorPanel when its **detachColorList:** method is invoked. Since NSColorPanel's list mode manages NSColorLists, this method need only be implemented by a custom color picker that manages NSColorLists itself.

Showing Opacity Controls

– (void)**alphaControlAddedOrRemoved:**(id)*sender* Sent by the color panel when the opacity controls have been hidden or displayed. If the color picker has its own opacity controls, it should hide or display them, depending on whether the sender’s **showsAlpha** method returns NO or YES.

Responding to a Resized View

– (void)**viewSizeChanged:**(id)*sender* Sent when the color picker’s superview has been resized in a way that might affect the color picker. *sender* is the NSColorPanel that contains the color picker.

NSDraggingDestination
(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Protocol Description

The NSDraggingDestination protocol declares methods that the destination (or recipient) of a dragged image must implement. The destination automatically receives NSDraggingDestination messages as an image enters, moves around inside, and then exits or is released within the destination's boundaries.

Note: In the text here and in the other dragging protocol descriptions, the term *dragging session* is the entire process during which an image is selected, dragged, released, and is absorbed or rejected by the destination. A *dragging operation* is the action that the destination takes in absorbing the image when it's released. The *dragging source* is the object that "owns" the image that's being dragged. It's specified as an argument to the **dragImage:...** message, sent to a NSWindow or NSView, that instigated the dragging session.

The Dragged Image

The image that's dragged in an image-dragging session is an NSImage object that represents data that's put on the pasteboard. Although a dragging destination can access the NSImage (through a method described in the NSDraggingInfo protocol), its primary concern is with the pasteboard data that the NSImage represents—the dragging operation that a destination ultimately performs is on the pasteboard data, not on the image itself.

Valid Destinations

Dragging is a visual phenomenon. To be an image-dragging destination, an object must represent a portion of screen real estate; thus, only NSWindows and NSViews can be destinations. Furthermore, you must announce the destination-candidacy of an NSWindow or NSView by sending it a **registerForDraggedTypes:** message. This method, defined in both classes, registers the pasteboard types that the object will accept. During a dragging session, a candidate destination will only receive NSDraggingDestination messages if the pasteboard types for which it is registered matches a type that's represented by the image that's being dragged.

Although NSDraggingDestination is declared as a protocol, the NSView and NSWindow subclasses that you create to adopt the protocol need only implement those methods that are pertinent. (The NSView and NSWindow classes provide private implementations for all of the methods.) In addition, an NSWindow or its delegate may implement these methods; the delegate's implementation takes precedent.

The Sender of Destination Messages

Each of the NSDraggingDestination methods sports a single argument: *sender*, the object that invoked the method. Within its implementations of the NSDraggingDestination methods, the destination can send NSDraggingInfo messages to *sender* to get more information on the current dragging session.

The Order of Destination Messages

The six NSDraggingDestination methods are invoked in a distinct order:

- As the image is dragged into the destination's boundaries, the destination is sent a **draggingEntered:** message.
- While the image remains within the destination, a series of **draggingUpdated:** messages are sent.
- If the image is dragged out of the destination, **draggingExited:** is sent and the sequence of NSDraggingDestination messages stops. If it re-enters, the sequence begins again (with a new **draggingEntered:** message).
- When the image is released, it either slides back to its source (and breaks the sequence) or a **prepareForDragOperation:** message is sent to the destination, depending on the value that was returned by the most recent invocation of **draggingEntered:** or **draggingUpdated:**.
- If the **prepareForDragOperation:** message returned YES, a **performDragOperation:** message is sent.
- Finally, if **performDragOperation:** returned YES, **concludeDragOperation:** is sent.

Before the Image is Released

– (NSDragOperation)**draggingEntered:**(id <NSDraggingInfo>)*sender*

Invoked when the dragged image enters the destination.

– (NSDragOperation)**draggingUpdated:**(id <NSDraggingInfo>)*sender*

Invoked periodically while the image is over the destination.

– (void)**draggingExited:**(id <NSDraggingInfo>)*sender*

Invoked when the dragged image exits the destination.

After the Image is Released

– (BOOL)**prepareForDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the image is released.

– (BOOL)**performDragOperation:**(id <NSDraggingInfo>)*sender*

Gives the destination an opportunity to perform the dragging operation.

– (void)**concludeDragOperation:**(id <NSDraggingInfo>)*sender*

Invoked when the dragging operation is complete.

NSDraggingInfo

Adopted By: no OpenStep classes

Declared In: AppKit/NSDragging.h

Protocol Description

The NSDraggingInfo protocol declares methods that supply information about a dragging session (see the NSDraggingDestination protocol, an informal protocol of NSObject, for definitions of dragging terms). A view or window first registers dragging types; it may then send NSDraggingInfo protocol messages while dragging occurs to get details about that dragging session.

NSDraggingInfo methods are designed to be invoked from within an object's implementation of the NSDraggingDestination protocol methods. An object that conforms to NSDraggingInfo is passed as the argument to each of the methods defined by NSDraggingDestination; NSDraggingInfo messages should be sent to this conforming object. The Application Kit supplies an NSDraggingInfo object automatically so that you never need to create a class that implements this protocol.

Dragging-Session Information

– (NSWindow *)**draggingDestinationWindow**	Returns the destination's Window.
– (NSPoint)**draggingLocation**	Returns the current location of the cursor's hot spot, reckoned in the base coordinate system of the destination object's Window.
– (NSPasteboard *)**draggingPasteboard**	Returns the Pasteboard that holds the dragged data.
– (int)**draggingSequenceNumber**	Returns a number that uniquely identifies the dragging session.
– (id)**draggingSource**	Returns the source, or "owner," of the dragged image. Returns **nil** if the source isn't in the same application as the destination.
– (NSDragOperation)**draggingSourceOperationMask**	Returns the operation mask declared by the source.

Image Information

– (NSImage *)**draggedImage**	Returns the image object that's being dragged. Don't invoke this method after the user has released the image, and don't release the object that this method returns.

– (NSPoint)**draggedImageLocation**

Returns the current location of the dragged image's origin. The image moves in lockstep with the cursor (the position of which is given by **draggingLocation**) but may be positioned at some offset. The point that's returned is reckoned in the base coordinate system of the destination object's Window.

Sliding the Image

– (void)**slideDraggedImageTo:**(NSPoint)*screenPoint*

Slides the image to the given location in the screen coordinate system. This method should only be invoked after the user has released the image but before it's removed from the screen.

NSDraggingSource
(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSDragging.h

Protocol Description

NSDraggingSource declares methods that can (or must) be implemented by the source object in a dragging session. (See the NSDraggingDestination protocol for definitions of dragging terms.) This *dragging source* is specified as an argument to the **dragImage:...** message, sent to a NSWindow or NSView, that instigated the dragging session.

Of the methods declared below, only the **draggingSourceOperationMaskForLocal:** method *must* be implemented. The other methods are invoked only if the dragging source implements them. All four methods are invoked automatically during a dragging session—you never send an NSDraggingSource message directly to an object.

Querying the Source

– (NSDragOperation)**draggingSourceOperationMaskForLocal:**(BOOL)*isLocal*
Returns a mask giving the operations that can be performed on the dragged image's data.

– (BOOL)**ignoreModifierKeysWhileDragging** Returns YES if modifier keys should have no effect on the type of operation performed.

Informing the Source

– (void)**draggedImage:**(NSImage *)*image* **beganAt:**(NSPoint)*screenPoint*
Invoked when the dragged image is displayed but before it starts following the mouse.

– (void)**draggedImage:**(NSImage *)*image* **endedAt:**(NSPoint)*screenPoint* **deposited:**(BOOL)*didDeposit*
Invoked after the dragged image has been released and the dragging destination has been given a chance to operate.

NSIgnoreMisspelledWords

Adopted By: NSText

Declared In: AppKit/NSSpellProtocol.h

Protocol Description

Implement this protocol to have the Ignore button in the Spelling panel function properly. The Ignore button allows the user to accept a word that the spelling checker believes is misspelled. In order for this action to update the "ignored words" list for the document being checked, the NSIgnoreMisspelledWords protocol must be implemented.

This protocol is necessary because a list of ignored words is useful only if it pertains to the entire document being checked, but the spelling checker (NSSpellChecker object) does not check the entire document for spelling at once. The spelling checker returns as soon as it finds a misspelled word. Thus, it checks only a subset of the document at any one time. The user usually wants to check the entire document, and so usually several spelling checks are run in succession until no misspelled words are found. This protocol allows the list of ignored words to be maintained per-document, even though the spelling checks are not run per-document.

The NSIgnoreMisspelledWords protocol specifies a method, **ignoreSpelling:**, which should be implemented like this:

```
– (void)ignoreSpelling:(id)sender
{
    [[NSSpellChecker sharedSpellChecker] ignoreWord:[[sender selectedCell] stringValue]
        inSpellDocumentWithTag:myDocumentTag];
}
```

The second argument to the NSSpellChecker method **ignoreWord:inSpellDocumentWithTag:** is a tag that the NSSpellChecker can use to distinguish the documents being checked. (See the discussion of "Matching a List of Ignored Words With the Document It Belongs To" in the description of the NSSpellChecker class.) Once the NSSpellChecker has a way to distinguish the various documents, it can append new ignored words to the appropriate list.

To make the ignored words feature useful, the application must store a document's ignored words list with the document. See the NSSpellChecker class description for more information.

Identifying the Source

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

Implement to allow an application to ignore misspelled words on a document-by-document basis. This message is sent by the NSSpellChecker instance to the object whose text is being checked. To inform the NSSpellChecker that a particular spelling should be ignored, the receiver asks the NSSpellChecker for the string value of its selected cell. It then sends the NSSpellChecker an **ignoreWord:inSpellDocumentWithTag:** message.

NSMenuActionResponder
(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSMenu.h

Protocol Description

This informal protocol allows your application to update the enabled or disabled status of an NSMenuCell. It declares only one method, **validateCell:**. By default, every time a user event occurs, NSMenu automatically enables and disables each visible menu cell based on criteria described later in this specification. Implement **validateCell:** in cases where you want to override NSMenu's default enabling scheme. This is described in more detail later.

There are two ways that NSMenuCells can be enabled or disabled: Explicitly, by sending the **setEnabled:** message, or automatically, as described below. NSMenuCells are updated automatically unless you send the message **setAutoenablesItems:NO** to the NSMenu object. You should never mix the two. That is, never use **setEnabled:** unless you have disabled the automatic updating.

Automatic Updating of NSMenuCells

Whenever a user event occurs, the NSMenu object updates the status of every visible menu cell. To update the status of a menu cell, NSMenu tries to find the object that responds to the NSMenuCell's action message. It searches the following objects in the following order until it finds one that responds to the action message.

- the NSMenuCell's target
- the key window's first responder
- the key window's delegate
- the main window's first responder
- the main window's delegate
- the NSApplication object
- the NSApplication's delegate
- the NSMenu's delegate

If none of these objects responds to the action message, the menu cell is disabled. If NSMenu finds an object that responds to the action message, it then checks to see if that object responds to the **validateCell:** message (the method defined in this informal protocol). If **validateCell:** is not implemented in that object, the menu cell is enabled. If it is implemented, the return value of **validateCell:** indicates whether the menu cell should be enabled or disabled.

For example, the NSText object implements the **copy:** method. If your application has a Copy menu cell that sends the **copy:** action message to the first responder, that menu cell is automatically enabled any time an NSText object is the first responder of the key or main window. If you have an object that might become the first responder and that object could allow users to select something that they aren't allowed to copy, you can implement the **validateCell:** method in that object. **validateCell:** can return NO if the forbidden items are selected and YES if they aren't. By implementing **validateCell:**, you can have the Copy menu item disabled even though its target object implements the **copy:** method. If instead your object *never* permits copying, then you would simply not implement **copy:** in that object, and the cell would be disabled automatically whenever the object is first responder.

If you send a **setEnabled:** message to enable or disable a menu cell when the automatic updating is turned on, other objects might reverse what you have done after another user event occurs. Using **setEnabled:**, you can never be sure that a menu cell is enabled or disabled or will remain that way. If your application must use **setEnabled:**, you must turn off the automatic enabling of menu cells (by sending **setAutoEnablesItems:NO** to NSMenu) in order to get predictable results.

Updating NSMenuCells

– (BOOL)**validateCell:**(id)*aCell*

Implemented to override the default action of updating an NSMenuCell. Return YES to enable the NSMenuCell, NO to disable it.

NSNibAwaking
(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSNibLoading.h

Protocol Description

This informal protocol consists of a single method, **awakeFromNib**. It's implemented to receive a notification message that's sent after objects have been loaded from an Interface Builder archive.

When **loadNibFile:owner:** or a related method loads an Interface Builder archive into an application, each custom object from the archive is first initialized with an **init** message (**initFrame:** if the object is a kind of View). Outlets are initialized via any **set***Variable***:** methods that are available (where *variable* is the name of an instance variable). (These methods are optional; the Objective C run time system automatically initializes outlets.) Finally, after all the objects are fully initialized, they each receive an **awakeFromNib** message.

The order in which objects are loaded from the archive is not guaranteed. Therefore, it's possible for a **set***Variable***:** message to be sent to an object before its companion objects have been unarchived. For this reason, **set***Variable***:** methods should not send messages to other objects in the archive. However, messages to other objects can safely be sent from within **awakeFromNib**—by this point it's assured that all the objects are unarchived and fully initialized.

Typically, **awakeFromNib** is implemented for only one object in the archive, the controlling or "owner" object for the other objects that are archived with it. For example, suppose that a nib file contained two Views that must be positioned relative to each other at run time. Trying to position them when either one of the Views is initialized (in a **set***Variable***:** method) might fail, since the other View might not be unarchived and initialized yet. However, it can be done in an **awakeFromNib** method:

```
- awakeFromNib
{
    NSRect viewFrame;

    [firstView getFrame:&viewFrame];
    [secondView moveTo:viewFrame.origin.x + someVariable
                    :viewFrame.origin.y];
    return self;
}
```

There's no default **awakeFromNib** method; an **awakeFromNib** message is only sent if an object implements it. The Application Kit declares a prototype for this method, but doesn't implement it.

Notification of Loading

– (void)**awakeFromNib**

Implemented to prepare an object for service after it has been loaded from an Interface Builder archive—a so-called "nib file". An **awakeFromNib** message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an **awakeFromNib** message, it's already guaranteed to have all its outlet instance variables set. There's no default **awakeFromNib** method.

NSServicesRequests
(informal protocol)

Category Of: NSObject

Declared In: AppKit/NSApplication.h

Protocol Description

This informal protocol consists of two methods, **writeSelectionToPasteboard:types:** and **readSelectionFromPasteboard:**. The first is implemented to provide data to a remote service, and the second to receive any data the remote service might send back. Both respond to messages that are generated when the user chooses a command from the Services menu.

Pasteboard Read/Write

– (BOOL)**readSelectionFromPasteboard:**(NSPasteboard *)*pboard*
Implemented to replace the current selection (that is, the text or objects that are currently selected) with data from *pboard.*

– (BOOL)**writeSelectionToPasteboard:**(NSPasteboard *)*pboard*
types:(NSArray *)*types*
Implemented to write the current selection to *pboard* as *types* data.

Application Kit Functions

Rectangle Drawing Functions

Optimize Drawing

void **NSEraseRect(**NSRect *aRect***)**	Erases the rectangle by filling it with white. (This does not alter the current drawing color.)
void **NSHighlightRect(**NSRect *aRect***)**	Highlights or unhighlights a rectangle by switching light gray for white and vice versa, when drawing on the screen. If not drawing to the screen, the rectangle is filled with light gray.
void **NSRectClip(**NSRect *aRect***)**	Intersects the current clipping path with the rectangle *aRect*, to determine a new clipping path.
void **NSRectClipList(**const NSRect **rects*, int *count***)**	Takes an array of *count* number of rectangles and intersects the current clipping path with each of them. Thus, the new clipping path is the graphic intersection of all the rectangles and the original clipping path.
void **NSRectFill(**NSRect *aRect***)**	Fills the rectangle referred to by *aRect* with the current color.
void **NSRectFillList(**const NSRect **rects*, int *count***)**	Fills an array of *count* rectangles with the current color.
void **NSRectFillListWithGrays(**const NSRect **rects*, const float **grays*, int *count***)**	Fills each rectangle in the array *rects* with the gray whose value is stored at the corresponding location in the array *grays*. Both arrays must be count elements long. Avoid rectangles that overlap, because the order in which they'll be filled can't be guaranteed.

Draw a Bordered Rectangle

void **NSDrawButton(**NSRect *aRect*, NSRect *clipRect***)**	Draws the bordered light gray rectangle whose appearance signifies a button in the OpenStep user interface. *aRect* is the bounds for the button, but only the area where *aRect* intersects *clipRect* is drawn.

void **NSDrawGrayBezel(**NSRect *aRect*, NSRect *clipRect***)**

Draws a bordered light gray rectangle with the appearance of a pushed-in button, clipped by intersecting with *clipRect*.

void **NSDrawGroove(**NSRect *aRect*, NSRect *clipRect***)**

Draws a light gray rectangle whose border is a groove, giving the appearance of a typical box in the OpenStep user interface.

NSRect **NSDrawTiledRects(**NSRect *boundsRect*, NSRect *clipRect*, const NSRectEdge **sides*, const float **grays*, int *count***)**

Draws an unfilled rectangle, clipped by *clipRect*, whose border is defined by the parallel arrays *sides* and *grays*, both of length *count*. Each element of *sides* specifies an edge of the rectangle, which is drawn with a width of 1.0 using the corresponding gray level from *grays*. If the *edges* array contains recurrences of the same edge, each is inset within the previous edge.

void **NSDrawWhiteBezel(**NSRect *aRect*, NSRect *clipRect***)**

Draws a white rectangle with a bezeled border. Only the area that intersects *clipRect* is drawn.

void **NSFrameRect(**NSRect *aRect***)**

Draws a frame of width 1.0 around the inside of a rectangle, using the current color.

void **NSFrameRectWithWidth(**NSRect *aRect*, float *frameWidth***)**

Draws a frame of width *frameWidth* around the inside of a rectangle, using the current color.

Color Functions

Get Information About Color Space and Window Depth

const NSWindowDepth ***NSAvailableWindowDepths(**void**)**

Returns a zero-terminated list of available window depths.

NSWindowDepth **NSBestDepth(**NSString **colorSpace*, int *bitsPerSample*, int *bitsPerPixel*, BOOL *planar*, BOOL **exactMatch***)**

Returns a window depth deep enough for the given number of colors, bits per sample, bits per pixel, and if planar. Upon return, the variable pointed to by *exactMatch* is YES if the window depth can accommodate all of the values given for all of the parameters, NO if not.

int **NSBitsPerPixelFromDepth(**NSWindowDepth *depth***)**

Returns the number of bits per pixel for the given window depth.

int **NSBitsPerSampleFromDepth(**NSWindowDepth *depth***)**

Returns the number of bits per sample (bits per pixel in each color component) for the given window depth.

NSString ***NSColorSpaceFromDepth(**NSWindowDepth *depth***)**

Returns the name of the color space that matches the given window depth.

int **NSNumberOfColorComponents(**NSString **colorSpaceName***)**

Returns the number of color components in the named color space.

BOOL **NSPlanarFromDepth(**NSWindowDepth *depth***)**

Returns YES if the given window depth is planar, NO if not.

Read the Color at a Screen Position

NSColor ***NSReadPixel(**NSPoint *location***)**

Returns the color of the pixel at the given location, which must be specified in the current view's coordinate system.

Text Functions

Filter Characters Entered into a Text Object

unsigned short **NSEditorFilter(**unsigned short *theChar*,
int *flags*,
NSStringEncoding *theEncoding***)**

Identical to **NSFieldFilter()** except that it passes on values corresponding to Return, Tab, and Shift-Tab directly to the NSText object.

unsigned short **NSFieldFilter(**unsigned short *theChar*,
int *flags*,
NSStringEncoding *theEncoding***)**

Checks each character the user types into an NSText object's text, allowing the user to move the selection among text fields by pressing Return, Tab, or Shift-Tab. Alphanumeric characters are passed to the NSText object for display. The function returns either the ASCII value of the character typed, 0 (for illegal characters or ones entered while a Command key is held down), or a constant that the Text object interprets as a movement command.

Calculate or Draw a Line of Text (in Text Object)

int **NSDrawALine(**id *self*,
NSLayInfo **layInfo***)**

Draws a line of text, using the global variables set by **NSScanALine()**. The return value has no significance.

int **NSScanALine(**id *self*,
NSLayInfo **layInfo***)**

Determines the placement of characters in a line of text. *self* refers to the NSText object calling the function, and **layInfo* is an NSLayInfo struct. The function returns 1 if a word's length exceeds the width of a line and the NSText's charWrap instance variable is NO. Otherwise, it returns 0.

Calculate Font Ascender, Descender, and Line Height (in Text Object)

void **NSTextFontInfo(**id *fid*,
float **ascender*, float **descender*,
float **lineHeight***)**

Calculates, and returns by reference, the ascender, descender, and line height values for the NSFont given by *font*.

Access Text Object's Word Tables

NSData * **NSDataWithWordTable(**const unsigned char **smartLeft*,
const unsigned char **smartRight*,
const unsigned char **charClasses*,
const NSFSM **wrapBreaks*,
int *wrapBreaksCount*,
const NSFSM **clickBreaks*,
int *clickBreaksCount*,
BOOL *charWrap***)**

Given pointers to word table structures, records the structures in the returned NSData object. The arguments are similar to those of **NSReadWordTable()**.

void **NSReadWordTable(**NSZone **zone*,
NSData **data*,
unsigned char ***smartLeft*,
unsigned char ***smartRight*,
unsigned char ***charClasses*,
NSFSM ***wrapBreaks*,
int **wrapBreaksCount*,
NSFSM ***clickBreaks*,
int **clickBreaksCount*,
BOOL **charWrap***)**

Given *data*, creates word tables in the memory zone specified by *zone*, returning (in the subsequent arguments) pointers to the various tables. The integer pointer arguments return the length of the preceding array, and *charWrap* indicates whether words whose length exceeds the NSText object's line length should be wrapped on a character-by-character basis.

Array Allocation Functions for Use by the NSText Class

NSTextChunk ***NSChunkCopy(**NSTextChunk **pc*, NSTextChunk **dpc***)**

Copies the array *pc* to the array *dpc* and returns a pointer to the copy.

NSTextChunk ***NSChunkGrow(**NSTextChunk **pc*, int *newUsed***)**

Increases the array identified by the pointer *pc* to a size of *newUsed* bytes.

NSTextChunk ***NSChunkMalloc(**int *growBy*, int *initUsed***)**

Allocates initial memory for a structure whose first field is an **NSTextChunk** structure and whose subsequent field is a variable-sized array. The amount of memory allocated is equal to *initUsed*. If *initUsed* is 0, *growBy* bytes are allocated. *growBy* specifies how much memory should be allocated when the chunk grows.

NSTextChunk ***NSChunkRealloc(**NSTextChunk **pc***)**

Increases the amount of memory available for the array identified by the pointer *pc*, as determined by the array's NSTextChunk.

NSTextChunk ***NSChunkZoneCopy(**NSTextChunk **pc*, NSTextChunk **dpc*, NSZone **zone***)**

Like **NSChunkCopy()**, but uses the specified zone of memory.

NSTextChunk ***NSChunkZoneGrow(**NSTextChunk **pc*, int *newUsed*, NSZone **zone***)**

Like **NSChunkGrow()**, but uses the specified zone of memory.

NSTextChunk ***NSChunkZoneMalloc(**int *growBy*, int *initUsed*, NSZone **zone***)**

Like **NSChunkMalloc()**, but uses the specified zone of memory.

NSTextChunk ***NSChunkZoneRealloc(**NSTextChunk **pc*, NSZone **zone***)**

Like **NSChunkRealloc()**, but uses the specified zone of memory.

Imaging Functions

Copy an image

void **NSCopyBitmapFromGState(**int *srcGstate*, NSRect *srcRect*, NSRect *destRect***)**

Copies the pixels in the rectangle *srcRect* to the rectangle *destRect*. The source rectangle is defined in the graphics state designated by *srcGstate*, and the destination is defined in the current graphics state.

void **NSCopyBits(**int *srcGstate*,
NSRect *srcRect*,
NSPoint *destPoint***)**

Copies the pixels in the rectangle *srcRect* to the location *destPoint*. The source rectangle is defined in the current graphics state if *srcGstate* is NSNullObject; otherwise, in the graphics state designated by *srcGstate*. The *destPoint* destination is defined in the current graphics state.

Render Bitmap Images

void **NSDrawBitmap**(NSRect *rect*,
int *pixelsWide*,
int *pixelsHigh*,
int *bitsPerSample*,
int *samplesPerPixel*,
int *bitsPerPixel*,
int *bytesPerRow*,
BOOL *isPlanar*,
BOOL *hasAlpha*,
NSString **colorSpaceName*,
const unsigned char *const *data*[5]**)**

Renders an image from a bitmap. *rect* is the rectangle in which the image is drawn, and *data* is the bitmap data, stored in up to 5 channels unless *isPlanar* is NO (in which case the channels are interleaved in a single array).

Attention Panel Functions

Create an Attention Panel without Running It Yet

id **NSGetAlertPanel(**NSString **title*,
NSString **msg*,
NSString **defaultButton*,
NSString **alternateButton*,
NSString **otherButton*, ...**)**

Returns an NSPanel object that you can use in a modal session. Unlike **NSRunAlertPanel()**, no button is displayed if *defaultButton* is NULL.

Create and Run an Attention Panel

int **NSRunAlertPanel(**NSString **title*,
NSString **msg*,
NSString **defaultButton*,
NSString **alternateButton*,
NSString **otherButton*, ...**)**

Creates an attention panel that alerts the user to some consequence of a requested action, and runs the panel in a modal event loop. *title* is the panel's title (by default, "Alert"); *msg* is the **printf()**-style message that's displayed in the panel; *defaultButton* (by default, "OK") is the title for the main button, also activated by Return; *alternateButton* and *otherButton* give two more choices, which are displayed only if the corresponding argument isn't NULL. The trailing arguments are a variable number of **printf()**-style arguments to msg.

int **NSRunLocalizedAlertPanel(**NSString **table*,
NSString **title*,
NSString **msg*,
NSString **defaultButton*,
NSString **alternateButton*,
NSString **otherButton*, ...**)**

Similar to **NSRunAlertPanel()**, but preferred, as it makes use of OpenStep's localization feature for languages of different countries.

Release an Attention Panel

void **NSReleaseAlertPanel(**id *panel***)**

Releases the specified alert panel.

Services Menu Functions

Determine Whether an Item Is Included in Services Menus

int **NSSetShowsServicesMenuItem(**NSString **item*,
BOOL *showService***)**

Determines (based on the value of *showService*) whether the *item* command will be included in other applications' Services menus. *item* describes a service provided by this application, and should be the same string entered in the "Menu Item:" field of the services file. The function returns 0 upon success.

BOOL **NSShowsServicesMenuItem(**NSString **item***)**

Returns YES if item is currently shown in Services menus.

Programmatically Invoke a Service

BOOL **NSPerformService(**NSString **item*, NSPasteboard **pboard***)**

Invokes a service found in the application's Services menu. *item* is the name of a Services menu item, in any language; a slash in this name represents a submenu. *pboard* must contain the data required by the service, and when the function returns, *pboard* will contain the data supplied by the service provider.

Force Services Menu to Update Based on New Ser vices

void **NSUpdateDynamicServices(**void**)**

Re-registers the services the application is willing to provide, by reading the file with the extension ".service" in the application path or in the standard path for services.

Other Application Kit Functions

Play the System Beep

void **NSBeep(**void**)**

Plays the system beep.

Return File-related Pasteboard Types

NSString ***NSCreateFileContentsPboardType(**NSString **fileType***)**

Returns a string naming a pasteboard type that represents a file's contents, based on the supplied string *fileType*. *fileType* should generally be the extension part of a file name. The conversion from a named file type to a pasteboard type is simple; no mapping to standard pasteboard types is attempted.

NSString ***NSCreateFilenamePboardType(**NSString **filename***)**

Returns a string naming a pasteboard type that represents a file name, based on the supplied string *filename*.

NSString ***NSGetFileType(**NSString **pboardType***)**

Returns the extension or file name from which the pasteboard type *pboardType* was derived. **nil** is returned if *pboardType* isn't a pasteboard type created by **NSCreateFileContentsPboardType()** or **NSCreateFilenamePboardType()**.

NSArray ***NSGetFileTypes(**NSArray **pboardTypes***)**

Accepts an array of pasteboard types and returns an array of the unique extensions and file names from the file-content and file-name types found in the input array. It returns **nil** if the input array contains no file-content or file-name types.

Draw a Distinctive Outline around Linked Data

void **NSFrameLinkRect(**NSRect *aRect*, BOOL *isDestination***)**

Draws a distinctive link outline just outside the rectangle *aRect*. To draw an outline around a destination link, *isDestination* should be YES, otherwise NO.

float **NSLinkFrameThickness(**void**)**

Returns the thickness of the link outline so that the outline can be properly erased by the application, or for other purposes.

Convert an Event Mask Type to a Mask

unsigned int **NSEventMaskFromType**(NSEventType *type*)

Returns the event mask corresponding to *type* (which is an enumeration constant). The returned mask equals 1 left-shifted by *type* bits.

Types and Constants

Application

id **NSApp**;	Represents the application's NSApplication object.
typedef struct _NSModalSession ***NSModalSession**;	This structure stores information used by the system during a modal session.

enum {
NSRunStoppedResponse,
NSRunAbortedResponse,
NSRunContinuesResponse
};
	Predefined return values for **runModalFor:** and **runModalSession:**.

NSString ***NSModalPanelRunLoopMode**;	Input-filter modes passed to NSRunLoop.
NSString ***NSEventTrackingRunLoopMode**;	

Box

typedef enum _NSTitlePosition { **NSNoTitle**, **NSAboveTop**, **NSAtTop**, **NSBelowTop**, **NSAboveBottom**, **NSAtBottom**, **NSBelowBottom** } **NSTitlePosition**;	This type's constants represent the locations where an NSBox's title is placed in relation to the border (**setTitlePosition:** and **titlePosition**).

Buttons

```
typedef enum _NSButtonType {
        NSMomentaryPushButton,
        NSPushOnPushOffButton,
        NSToggleButton,
        NSSwitchButton,
        NSRadioButton,
        NSMomentaryChangeButton,
        NSOnOffButton
} NSButtonType;
```

The constants of **NSButtonType** indicate the way NSButtons and NSButtonCells behave when pressed, and how they display their state. They are used in NSButton's **setType:** method.

Cells and Button Cells

```
typedef enum _NSCellType {
        NSNullCellType,
        NSTextCellType,
        NSImageCellType
} NSCellType;
```

Represent different types of NSCell objects.
- **NSNullCellType**: No display.
- **NSTextCellType**: Displays text.
- **NSImageCellType**: Displays an image.
- **NSCellType**: Returned from **type** and set via **setType:**.

```
typedef enum _NSCellImagePosition {
        NSNoImage,
        NSImageOnly,
        NSImageLeft,
        NSImageRight,
        NSImageBelow,
        NSImageAbove,
        NSImageOverlaps
} NSCellImagePosition;
```

Represent the position of an NSButtonCell relative to its title. Returned from **imagePosition** and set through **setImagePosition:**.

typedef enum _NSCellAttribute {
    NSCellDisabled,
    NSCellState,
    NSPushInCell,
    NSCellEditable,
    NSChangeGrayCell,
    NSCellHighlighted,
    NSCellLightsByContents,
    NSCellLightsByGray,
    NSChangeBackgroundCell,
    NSCellLightsByBackground,
    NSCellIsBordered,
    NSCellHasOverlappingImage,
    NSCellHasImageHorizontal,
    NSCellHasImageOnLeftOrBottom,
    NSCellChangesContents,
    NSCellIsInsetButton
} **NSCellAttribute**;
	The constant values of **NSCellAttribute** represent parameters that you can set and access through NSCell's and NSButtonCell's **setParameter:to:** and **getParameter:** methods. Only the first five constants are used by NSCell; the others apply to NSButtonCells only.

enum {
    NSAnyType,
    NSIntType,
    NSPositiveIntType,
    NSFloatType,
    NSPositiveFloatType,
    NSDateType,
    NSDoubleType,
    NSPositiveDoubleType
};
	Numeric data types that an NSCell can accept. Used as the argument for **setEntryType:**.

enum {
    NSNoCellMask,
    NSContentsCellMask,
    NSPushInCellMask,
    NSChangeGrayCellMask,
    NSChangeBackgroundCellMask
};
	NSButtonCell uses these values to determine how to highlight a button cell or show an ON state (returned/passed in **showsStateBy**/**setShowsStateBy** and **highlightsBy**/**setHighlightsBy**).

Color

enum {
    NSGrayModeColorPanel,
    NSRGBModeColorPanel,
    NSCMYKModeColorPanel,
    NSHSBModeColorPanel,
    NSCustomPaletteModeColorPanel,
    NSColorListModeColorPanel,
    NSWheelModeColorPanel
};
	Tags that identify modes (or views) in the color panel.

enum {
    NSColorPanelGrayModeMask,
    NSColorPanelRGBModeMask,
    NSColorPanelCMYKModeMask,
    NSColorPanelHSBModeMask,
    NSColorPanelCustomPaletteModeMask,
    NSColorPanelColorListModeMask,
    NSColorPanelWheelModeMask,
    NSColorPanelAllModesMask
};
	Bit masks for determining the current mode (or view) of the color panel.

Data Link

typedef int **NSDataLinkNumber**;	Returned by NSDataLink's **linkNumber** method as a persistent identifier of a destination link.
NSString ***NSDataLinkFileNameExtension**;	The file name suffix to be used when data links are saved. The default is **objlink**.

typedef enum _NSDataLinkDisposition {
    NSLinkInDestination,
    NSLinkInSource,
    NSLinkBroken
} **NSDataLinkDisposition**;
	Returned by NSDataLink's **disposition** method to identify a link as a destination link, a source link, or a broken link.

typedef enum _NSDataLinkUpdateMode {
    NSUpdateContinuously,
    NSUpdateWhenSourceSaved,
    NSUpdateManually,
    NSUpdateNever
} **NSDataLinkUpdateMode**;
	Identifies when a link's data is to be updated. Set through the **setUpdateMode:** method and returned by **updateMode**.

Drag Operation

```
typedef enum _NSDragOperation {
        NSDragOperationNone,
        NSDragOperationCopy,
        NSDragOperationLink,
        NSDragOperationGeneric,
        NSDragOperationPrivate,
        NSDragOperationAll
} NSDragOperation;
```

The constants of this type identify different kinds of dragging operations. **NSDragOperationNone** implies that the operation is rejected. **NSDragOperationPrivate** means that the system leaves the cursor alone.

Event Handling

typedef enum _NSEventType {
NSLeftMouseDown,
NSLeftMouseUp,
NSRightMouseDown,
NSRightMouseUp,
NSMouseMoved,
NSLeftMouseDragged,
NSRightMouseDragged,
NSMouseEntered,
NSMouseExited,
NSKeyDown,
NSKeyUp,
NSFlagsChanged,
NSPeriodic,
NSCursorUpdate
} **NSEventType**;

Each constant of **NSEventType** identifies an event type. (See the NSEvent class description.)

enum {
NSUpArrowFunctionKey = 0xF700,
NSDownArrowFunctionKey = 0xF701,
NSLeftArrowFunctionKey = 0xF702,
NSRightArrowFunctionKey = 0xF703,
NSF1FunctionKey = 0xF704,
NSF2FunctionKey = 0xF705,
NSF3FunctionKey = 0xF706,
NSF4FunctionKey = 0xF707,
NSF5FunctionKey = 0xF708,
NSF6FunctionKey = 0xF709,
NSF7FunctionKey = 0xF70A,
NSF8FunctionKey = 0xF70B,
NSF9FunctionKey = 0xF70C,
NSF10FunctionKey = 0xF70D,
NSF11FunctionKey = 0xF70E,
NSF12FunctionKey = 0xF70F,
NSF13FunctionKey = 0xF710,
NSF14FunctionKey = 0xF711,
NSF15FunctionKey = 0xF712,
NSF16FunctionKey = 0xF713,
NSF17FunctionKey = 0xF714,
NSF18FunctionKey = 0xF715,
NSF19FunctionKey = 0xF716,
NSF20FunctionKey = 0xF717,
NSF21FunctionKey = 0xF718,

Unicodes that identify function keys on the keyboard, OpenStep reserves the range 0xF700-0xF8FF for this purpose. The availability of some keys is system-dependent.

NSF22FunctionKey = 0xF719,
NSF23FunctionKey = 0xF71A,
NSF24FunctionKey = 0xF71B,
NSF25FunctionKey = 0xF71C,
NSF26FunctionKey = 0xF71D,
NSF27FunctionKey = 0xF71E,
NSF28FunctionKey = 0xF71F,
NSF29FunctionKey = 0xF720,
NSF30FunctionKey = 0xF721,
NSF31FunctionKey = 0xF722,
NSF32FunctionKey = 0xF723,
NSF33FunctionKey = 0xF724,
NSF34FunctionKey = 0xF725,
NSF35FunctionKey = 0xF726,
NSInsertFunctionKey = 0xF727,
NSDeleteFunctionKey = 0xF728,
NSHomeFunctionKey = 0xF729,
NSBeginFunctionKey = 0xF72A,
NSEndFunctionKey = 0xF72B,
NSPageUpFunctionKey = 0xF72C,
NSPageDownFunctionKey = 0xF72D,
NSPrintScreenFunctionKey = 0xF72E,
NSScrollLockFunctionKey = 0xF72F,
NSPauseFunctionKey = 0xF730,
NSSysReqFunctionKey = 0xF731,
NSBreakFunctionKey = 0xF732,
NSResetFunctionKey = 0xF733,
NSStopFunctionKey = 0xF734,
NSMenuFunctionKey = 0xF735,
NSUserFunctionKey = 0xF736,
NSSystemFunctionKey = 0xF737,
NSPrintFunctionKey = 0xF738,
NSClearLineFunctionKey = 0xF739,
NSClearDisplayFunctionKey = 0xF73A,
NSInsertLineFunctionKey = 0xF73B,
NSDeleteLineFunctionKey = 0xF73C,
NSInsertCharFunctionKey = 0xF73D,
NSDeleteCharFunctionKey = 0xF73E,
NSPrevFunctionKey = 0xF73F,
NSNextFunctionKey = 0xF740,
NSSelectFunctionKey = 0xF741,
NSExecuteFunctionKey = 0xF742,
NSUndoFunctionKey = 0xF743,
NSRedoFunctionKey = 0xF744,
NSFindFunctionKey = 0xF745,
NSHelpFunctionKey = 0xF746,

NSModeSwitchFunctionKey = 0xF747
};

enum {
NSAlphaShiftKeyMask,
NSShiftKeyMask,
NSControlKeyMask,
NSAlternateKeyMask,
NSCommandKeyMask,
NSNumericPadKeyMask,
NSHelpKeyMask,
NSFunctionKeyMask
};

Device-independent bit masks for evaluating event-modifier flags to determine which modifier key (if any) was pressed.

enum {
NSLeftMouseDownMask,
NSLeftMouseUpMask,
NSRightMouseDownMask,
NSRightMouseUpMask,
NSMouseMovedMask,
NSLeftMouseDraggedMask,
NSRightMouseDraggedMask,
NSMouseEnteredMask,
NSMouseExitedMask,
NSKeyDownMask,
NSKeyUpMask,
NSFlagsChangedMask,
NSPeriodicMask,
NSCursorUpdateMask,
NSAnyEventMask
};

Bit masks for determining the type of events.

Exceptions

Global Exception Strings

The following global strings identify the exceptions returned by various operations in the Application Kit. They are defined in NSErrors.h.

NSString ***NSAbortModalException**;

NSString ***NSAbortPrintingException**;

NSString ***NSAppKitIgnoredException**;

NSString ***NSAppKitVirtualMemoryException**;

NSString ***NSBadBitmapParametersException**;

NSString ***NSBadComparisonException**;

NSString ***NSBadRTFColorTableException**;

NSString ***NSBadRTFDirectiveException**;

NSString ***NSBadRTFFontTableException**;

NSString ***NSBadRTFStyleSheetException**;

NSString ***NSBrowserIllegalDelegateException**;

NSString ***NSColorListIOException**;

NSString ***NSColorListNotEditableException**;

NSString ***NSDraggingException**;

NSString ***NSFontUnavailableException**;

NSString ***NSIllegalSelectorException**;

NSString ***NSImageCacheException**;

NSString ***NSNibLoadingException**;

NSString ***NSPPDIncludeNotFoundException**;

NSString ***NSPPDIncludeStackOverflowException**;

NSString ***NSPPDIncludeStackUnderflowException**;

NSString ***NSPPDParseException**;

NSString ***NSPasteboardCommunicationException**;

NSString ***NSPrintOperationExistsException**; (Defined in NSPrintOperation.h.)

NSString ***NSPrintPackageException**;

NSString ***NSPrintingCommunicationException**;

NSString ***NSRTFPropertyStackOverflowException**;

NSString ***NSTIFFException**;

NSString ***NSTextLineTooLongException**;

NSString ***NSTextNoSelectionException**;

NSString ***NSTextReadException**;

NSString ***NSTextWriteException**;

NSString ***NSTypedStreamVersionException**;

NSString ***NSWindowServerCommunicationException**;

NSString ***NSWordTablesReadException**;

NSString ***NSWordTablesWriteException**;

Fonts

typedef unsigned int **NSFontTraitMask**;	Characterizes one or more of a font's traits. It's used as an argument type for several of the methods in the NSFontManager class. You build a mask by OR'ing together the following enumeration constants.

enum {
NSItalicFontMask,
NSBoldFontMask,
NSUnboldFontMask,
NSNonStandardCharacterSetFontMask,
NSNarrowFontMask,
NSExpandedFontMask,
NSCondensedFontMask,
NSSmallCapsFontMask,
NSPosterFontMask,
NSCompressedFontMask,
NSUnitalicFontMask
};
	Values used by NSFontManager to identify font traits.

typedef unsigned int **NSGlyph**;	A type for numbers identifying font glyphs. It's used as the argument type for several of the methods in NSFont.

enum {
NSFPPreviewButton ,
NSFPRevertButton,
NSFPSetButton,
NSFPPreviewField,
NSFPSizeField,
NSFPSizeTitle,
NSFPCurrentField
};
	Tags identifying views in the font panel.

const float *NSFontIdentityMatrix;	Identifies a font matrix that's used for fonts displayed in an NSView object that has an unflipped coordinate system.
NSString *NSAFMAscender; NSString *NSAFMCapHeight; NSString *NSAFMCharacterSet; NSString *NSAFMDescender; NSString *NSAFMEncodingScheme; NSString *NSAFMFamilyName; NSString *NSAFMFontName; NSString *NSAFMFormatVersion; NSString *NSAFMFullName; NSString *NSAFMItalicAngle; NSString *NSAFMMappingScheme; NSString *NSAFMNotice; NSString *NSAFMUnderlinePosition; NSString *NSAFMUnderlineThickness; NSString *NSAFMVersion; NSString *NSAFMWeight; NSString *NSAFMXHeight;	Global keys to access the values available in the AFM dictionary. You can convert the appropriate values (e.g., ascender, cap height) to floating point values by using NSString's floatValue method.

Graphics

typedef int NSWindowDepth	This type gives the window-depth limit. Use the NSAvailableWindowDepths() function to get a list of available window depths. Use the functions NSBitsPerSampleFromDepth(), NSBitsPerPixelFromDepth(), NSPlanarFromDepth, and NSColorSpaceFromDepth() to extract information from a window depth. The NSWindowDepth type is also used as an argument type of methods in NSScreen and NSWindow.

typedef enum _NSTIFFCompression {
    NSTIFFCompressionNone = 1,
    NSTIFFCompressionCCITTFAX3 = 3,
    NSTIFFCompressionCCITTFAX4 = 4,
    NSTIFFCompressionLZW = 5,
    NSTIFFCompressionJPEG = 6,
    NSTIFFCompressionNEXT = 32766,
    NSTIFFCompressionPackBits = 32773,
    NSTIFFCompressionOldJPEG = 32865
} NSTIFFCompression;
	The constants defined in this type represent the various TIFF (tag image file format) data compression schemes. They are defined in NSBitMapImageRep and used in several methods of that class as well as in the TIFFRepresentationUsingCompression:factor: method of NSImage.

```
enum {
    NSImageRepMatchesDevice
};
```

NSImageRepMatchesDevice indicates that the value varies according to the output device. It can be passed in (or received back) as the value of NSImageRep's **bitsPerSample**, **pixelsWide**, and **pixelsHigh**.

Colorspace Names

Predefined colorspace names. Used as arguments in **NSDrawBitMap()** and **NSNumberOfColorComponents()**; value returned from **NSColorSpaceFromDepth()**.

NSString ***NSCalibratedWhiteColorSpace**;

NSString ***NSCalibratedBlackColorSpace**;

NSString ***NSCalibratedRGBColorSpace**;

NSString ***NSDeviceWhiteColorSpace**;

NSString ***NSDeviceBlackColorSpace**;

NSString ***NSDeviceRGBColorSpace**;

NSString ***NSDeviceCMYKColorSpace**;

NSString ***NSNamedColorSpace**;

NSString ***NSCustomColorSpace**;

Gray Values

Standard gray values for the 2-bit deep grayscale colorspace.

const float **NSBlack**;

const float **NSDarkGray**;

const float **NSWhite**;

const float **NSLightGray**;

Device Dictionary Keys

Keys to get designated values from device dictionaries.

NSString ***NSDeviceResolution**;

NSString ***NSDeviceColorSpaceName**

NSString ***NSDeviceBitsPerSample**;

NSString ***NSDeviceIsScreen**;

NSString ***NSDeviceIsPrinter**;

NSString ***NSDeviceSize**;

Matrix

typedef enum _NSMatrixMode {
    NSRadioModeMatrix,
    NSHighlightModeMatrix,
    NSListModeMatrix,
    NSTrackModeMatrix
} **NSMatrixMode**;

The constants in this type represent the modes of operation of an NSMatrix.

Notifications

Notifications are posted to all interested observers of a specific condition to alert them that the condition has occurred. Global strings contain the actual text of the notification. In the Application Kit, these are defined per class. See the Foundation's NSNotification and NSNotificationCenter for details.

NSString ***NSApplicationDidBecomeActiveNotification**;

NSApplication

NSString ***NSApplicationDidFinishLaunchingNotification**;

NSString ***NSApplicationDidHideNotification**;

NSString ***NSApplicationDidResignActiveNotification**;

NSString ***NSApplicationDidUnhideNotification**;

NSString ***NSApplicationDidUpdateNotification**;

NSString ***NSApplicationWillBecomeActiveNotification**;

NSString ***NSApplicationWillFinishLaunchingNotification**;

NSString ***NSApplicationWillHideNotification**;

NSString ***NSApplicationWillResignActiveNotification**;

NSString ***NSApplicationWillUnhideNotification**;

NSString ***NSApplicationWillUpdateNotification**;

NSString ***NSColorListChangedNotification**;	NSColorList
NSString ***NSColorPanelColorChangedNotification**;	NSColorPanel
NSString ***NSControlTextDidBeginEditingNotification**;	NSControl
NSString ***NSControlTextDidEndEditingNotification**;	
NSString ***NSControlTextDidChangeNotification**;	
NSString ***NSImageRepRegistryChangedNotification**;	NSImageRep
NSString ***NSSplitViewDidResizeSubviewsNotification**;	NSSplitView
NSString ***NSSplitViewWillResizeSubviewsNotification**;	
NSString ***NSTextDidBeginEditingNotification;**	NSText
NSString ***NSTextDidEndEditingNotification;**	
NSString ***NSTextDidChangeNotification;**	
NSString ***NSViewFrameChangedNotification**;	NSView
NSString ***NSViewFocusChangedNotification**;	
NSString ***NSWindowDidBecomeKeyNotification**;	NSWindow
NSString ***NSWindowDidBecomeMainNotification**;	
NSString ***NSWindowDidChangeScreenNotification**;	
NSString ***NSWindowDidDeminiaturizeNotification**;	
NSString ***NSWindowDidExposeNotification**;	
NSString ***NSWindowDidMiniaturizeNotification**;	
NSString ***NSWindowDidMoveNotification**;	
NSString ***NSWindowDidResignKeyNotification**;	
NSString ***NSWindowDidResignMainNotification**;	

NSString ***NSWindowDidResizeNotification**;

NSString ***NSWindowDidUpdateNotification**;

NSString ***NSWindowWillCloseNotification**;

NSString ***NSWindowWillMiniaturizeNotification**;

NSString ***NSWindowWillMoveNotification**;

NSString ***NSWorkspaceDidLaunchApplicationNotification**; NSWorkspace

NSString ***NSWorkspaceDidMountNotification**;

NSString ***NSWorkspaceDidPerformFileOperationNotification**;

NSString ***NSWorkspaceDidTerminateApplicationNotification**;

NSString ***NSWorkspaceDidUnmountNotification**;

NSString ***NSWorkspaceWillLaunchApplicationNotification**;

NSString ***NSWorkspaceWillPowerOffNotification**;

NSString ***NSWorkspaceWillUnmountNotification**;

Panel

enum {
    NSOKButton = 1,
    NSCancelButton = 0
};

Values returned by the standard panel buttons, OK and Cancel.

enum {
    NSAlertDefaultReturn = 1,
    NSAlertAlternateReturn = 0,
    NSAlertOtherReturn = –1,
    NSAlertErrorReturn = –2
};

Values returned by the **NSRunAlertPanel()** function and by **runModalSession:** when the modal session is run with a Panel provided by **NSGetAlertPanel()**.

Page Layout

enum {
NSPLImageButton,
NSPLTitleField,
NSPLPaperNameButton,
NSPLUnitsButton,
NSPLWidthForm,
NSPLHeightForm,
NSPLOrientationMatrix,
NSPLCancelButton,
NSPLOKButton
};

Tags that identify buttons, fields, and other views of the Page Layout panel.

Pasteboard

Pasteboard Type Globals

Identifies the standard pasteboard types. These are used in a variety of NSPasteboard methods and functions.

NSString ***NSStringPboardType**;

NSString ***NSColorPboardType**;

NSString ***NSFileContentsPboardType**;

NSString ***NSFilenamesPboardType**;

NSString ***NSFontPboardType**;

NSString ***NSRulerPboardType**;

NSString ***NSPostScriptPboardType**;

NSString ***NSTabularTextPboardType**;

NSString ***NSRTFPboardType**;

NSString ***NSTIFFPboardType**;

NSString ***NSDataLinkPboardType**; (Defined in NSDataLink.h.)

NSString ***NSGeneralPboardType**; (Defined in NSSelection.h.)

Pasteboard Name Globals

Identifies the standard pasteboard names. Used in class method **pasteboardWithName:** to get a pasteboard by name.

NSString *__NSDragPboard__;

NSString *__NSFindPboard__;

NSString *__NSFontPboard__;

NSString *__NSGeneralPboard__;

NSString *__NSRulerPboard__;

Printing

typedef enum _NSPrinterTableStatus {
    NSPrinterTableOK,
    NSPrinterTableNotFound,
    NSPrinterTableError
} **NSPrinterTableStatus**;

These constants describe the state of a printer-information table stored by an NSPrinter object. It is the argument type of the return value of **statusForTable:**.

typedef enum _NSPrintingOrientation {
    NSPortraitOrientation,
    NSLandscapeOrientation
} **NSPrintingOrientation**;

These constants represent the way a page is oriented for printing.

typedef enum _NSPrintingPageOrder {
    NSDescendingPageOrder,
    NSSpecialPageOrder,
    NSAscendingPageOrder,
    NSUnknownPageOrder
} **NSPrintingPageOrder**;

These constants describe the order in which pages are spooled for printing. **NSSpecialPageOrder** tells the spooler not to rearrange pages. Set through NSPrintingOperation's **setPageOrder:** method and returned by its **pageOrder** method.

typedef enum _NSPrintingPaginationMode {
    NSAutoPagination,
    NSFitPagination,
    NSClipPagination
} **NSPrintingPaginationMode**;

These constants represent the different ways an image is divided into pages during pagination. Pagination can occur automatically, the image can be forced onto a page, or it can be clipped to a page.

```
enum {
    NSPPSaveButton,
    NSPPPreviewButton,
    NSFaxButton,
    NSPPTitleField,
    NSPPImageButton,
    NSPPNameTitle,
    NSPPNameField,
    NSPPNoteTitle,
    NSPPNoteField,
    NSPPStatusTitle,
    NSPPStatusField,
    NSPPCopiesField,
    NSPPPageChoiceMatrix,
    NSPPPageRangeFrom,
    NSPPPageRangeTo,
    NSPPScaleField,
    NSPPOptionsButton,
    NSPPPaperFeedButton,
    NSPPLayoutButton
};
```

Tags that identify text fields, controls, and other views in the Print Panel.

Printing Information Dictionary Keys

The keys in the mutable dictionary associated with NSPrintingInfo. See NSPrintingInfo.h for types and descriptions of values.

NSString ***NSPrintAllPages**;

NSString ***NSPrintBottomMargin**;

NSString ***NSPrintCopies**;

NSString ***NSPrintFaxCoverSheetName**;

NSString ***NSPrintFaxHighResolution**;

NSString ***NSPrintFaxModem**;

NSString ***NSPrintFaxReceiverNames**;

NSString ***NSPrintFaxReceiverNumbers**;

NSString ***NSPrintFaxReturnReceipt**;

NSString ***NSPrintFaxSendTime**;

NSString ***NSPrintFaxTrimPageEnds**;

NSString ***NSPrintFaxUseCoverSheet**;

NSString ***NSPrintFirstPage**;

NSString ***NSPrintHorizonalPagination**;

NSString ***NSPrintHorizontallyCentered**;

NSString ***NSPrintJobDisposition**;

NSString ***NSPrintJobFeatures**;

NSString ***NSPrintLastPage**;

NSString ***NSPrintLeftMargin**;

NSString ***NSPrintManualFeed**;

NSString ***NSPrintOrientation**;

NSString ***NSPrintPackageException**;

NSString ***NSPrintPagesPerSheet**;

NSString ***NSPrintPaperFeed**;

NSString ***NSPrintPaperName**;

NSString ***NSPrintPaperSize**;

NSString ***NSPrintPrinter**;

NSString ***NSPrintReversePageOrder**;

NSString ***NSPrintRightMargin**;

NSString ***NSPrintSavePath**;

NSString ***NSPrintScalingFactor**;

NSString ***NSPrintTopMargin**;

NSString ***NSPrintVerticalPagination**;

NSString ***NSPrintVerticallyCentered**;

Print Job Disposition Values

These global constants define the disposition of a print job. See NSPrintInfo's **setJobDisposition:** and **jobDisposition**.

NSString ***NSPrintCancelJob**;

NSString ***NSPrintFaxJob**;

NSString ***NSPrintPreviewJob**;

NSString *NSPrintSaveJob;

NSString *NSPrintSpoolJob;

Save Panel

```
enum {
    NSFileHandlingPanelImageButton,
    NSFileHandlingPanelTitleField,
    NSFileHandlingPanelBrowser,
    NSFileHandlingPanelCancelButton,
    NSFileHandlingPanelOKButton,
    NSFileHandlingPanelForm,
    NSFileHandlingPanelHomeButton,
    NSFileHandlingPanelDiskButton,
    NSFileHandlingPanelDiskEjectButton
};
```

Tags that identify buttons, fields, and other views in the Save Panel.

Scroller

```
typedef enum _NSScrollArrowPosition {
    NSScrollerArrowsMaxEnd,
    NSScrollerArrowsMinEnd,
    NSScrollerArrowsNone
} NSScrollArrowPosition;
```

NSScroller uses these constants in its **setArrowPosition:** method to set the position of the arrows within the scroller.

```
typedef enum _NSScrollerPart {
    NSScrollerNoPart,
    NSScrollerDecrementPage,
    NSScrollerKnob,
    NSScrollerIncrementPage,
    NSScrollerDecrementLine,
    NSScrollerIncrementLine,
    NSScrollerKnobSlot
} NSScrollerPart;
```

NSScroller uses these constants in its **hitPart** method to identify the part of the scroller specified in a mouse event.

```
typedef enum _NSScrollerUsablePart {
    NSNoScrollerParts,
    NSOnlyScrollerArrows,
    NSAllScrollerParts
} NSUsableScrollerParts;
```

These constants define the usable parts of an NSScroller object.

typedef enum _NSScrollerArrow {
    NSScrollerIncrementArrow,
    NSScrollerDecrementArrow
} **NSScrollerArrow**;

These constants indicate the two types of scroller arrow. NSScroller's **drawArrow:highlight:** method takes an NSScrollerArrow as the first argument.

const float **NSScrollerWidth**;

Identifies the default width of a vertical NSScroller object and the default height of a horizontal NSScroller object.

Text

typedef struct _NSBreakArray {
    NSTextChunk **chunk**;
    NSLineDesc **breaks**[1];
} **NSBreakArray**;

Holds line-break information for an NSText object. It's mainly an array of line descriptors.

typedef struct _NSCharArray {
    NSTextChunk **chunk**;
    unsigned char **text**[1];
} **NSCharArray**;

Holds the character array for the current line in the NSText object.

typedef unsigned short (***NSCharFilterFunc**) (
    unsigned short *charCode*,
    int *flags*,
    NSStringEncoding *theEncoding*);

The character filter function analyzes each character the user enters in the NSText object.

typedef struct _NSFSM {
    const struct _NSFSM ***next**;
    short **delta**;
    short **token**;
} **NSFSM**;

A word definition finite-state machine structure used by an NSText object.

typedef struct _NSHeightChange {
    NSLineDesc **lineDesc**;
    NSHeightInfo **heightInfo**;
} **NSHeightChange**;

Associates line descriptors and line-height information in an NSText object.

typedef struct _NSHeightInfo {
    float **newHeight**;
    float **oldHeight**;
    NSLineDesc **lineDesc**;
} **NSHeightInfo**;

Stores height information for each line of text in an NSText object.

```
typedef struct _NSLay {
        float x;
        float y;
        short   offset;
        short   chars;
        id  font;
        void   *paraStyle;
        NSRun *run;
        NSLayFlags lFlags;
} NSLay;
```

Represents a single sequence of text in a line and records everything needed to select or draw that piece.

```
typedef struct _NSLayArray {
        NSTextChunk chunk;
        NSLay   lays[1];
} NSLayArray;
```

Holds the layout for the current line. Since the structure's first field is an **NSTextChunk** structure, **NSLayArray**s can be manipulated by the functions that manage variable-sized arrays of records.

```
typedef struct {
        unsigned int mustMove:1;
        unsigned int isMoveChar:1;
        unsigned int RESERVED:14;
} NSLayFlags;
```

Records whether a text lay in an NSText object needs special treatment (e.g., because of non-printing characters).

```
typedef struct _NSLayInfo {
        NSRect rect;
        float descent;
        float width;
        float left;
        float right;
        float rightIndent;
        NSLayArray *lays;
        NSWidthArray *widths;
        NSCharArray *chars;
        NSTextCache cache;
        NSRect *textClipRect;
        struct _lFlags {
            unsigned int horizCanGrow:1;
            unsigned int vertCanGrow:1;
            unsigned int erase:1;
            unsigned int ping:1;
            unsigned int endsParagraph:1;
            unsigned int resetCache:1;
            unsigned int RESERVED:10;
        } lFlags;
} NSLayInfo;
```

NSText's scanning and drawing functions use this structure to communicate information about lines of text.

```
typedef short NSLineDesc;
```

Used to identify lines of text in the NSText object.

```
typedef enum _NSParagraphProperty {
        NSLeftAlignedParagraph,
        NSRightAlignedParagraph,
        NSCenterAlignedParagraph,
        NSJustificationAlignedParagraph,
        NSFirstIndentParagraph,
        NSIndentParagraph,
        NSAddTabParagraph,
        NSRemoveTabParagraph,
        NSLeftMarginParagraph,
        NSRightMarginParagraph
} NSParagraphProperty;
```

The constants of this type identify specific paragraph properties for selected text. NSText's s**etSelProp:** method takes this argument type.

```
typedef struct _NSRun {
        id  font;
        int chars;
        void   *paraStyle;
        int  textRGBColor;
        unsigned char   superscript;
        unsigned char   subscript;
        id   info;
        NSRunFlags rFlags;
} NSRun;
```

In an NSText object, this structure represents a single sequence of text with a given format.

```
typedef struct _NSRunArray {
        NSTextChunk chunk;
        NSRun   runs[1];
} NSRunArray;
```

This structure holds the array of text runs in an NSText object. Since the first field is an NSTextChunk structure you can manipulate the items in the array with the functions that manage variable-sized arrays of records.

```
typedef struct {
        unsigned int underline:1;
        unsigned int dummy:1;
        unsigned int subclassWantsRTF:1;
        unsigned int graphic:1;
        unsigned int forcedSymbol:1;
        unsigned int RESERVED:11;
} NSRunFlags;
```

The fields of this structure record whether a run in an NSText object contains graphics, is underlined, or if an alternate character forced the use of a symbol.

Represents one end of a selection in an NSText object.

typedef struct _NSSelPt {	
int **cp**;	Character position.
int **line**;	Offset of LineDesc in break table.
float **x**;	x coordinate.
float **y**;	y coordinate.
int **c1st**;	Character position of first character in the line.
float **ht**;	Line height.
} **NSSelPt**;	

typedef struct _NSTabStop {
    short **kind**;
    float **x**;
} **NSTabStop**;
	This structure describes an NSText object's tab stops.

typedef struct _NSTextBlock {
    struct _NSTextBlock ***next**;
    struct _NSTextBlock ***prior**;
    struct _tbFlags {
        unsigned int **malloced**:1;
        unsigned int **PAD**:15;
    } **tbFlags**;
    short **chars**;
    unsigned char ***text**;
} **NSTextBlock**;
	A structure holds text characters in blocks no bigger than **NSTextBlockSize** (see below). A linked list of these text blocks comprises the text for an NSText object.

typedef struct _NSTextCache {
    int **curPos**;
    NSRun ***curRun**;
    int **runFirstPos**;
    NSTextBlock ***curBlock**;
    int **blockFirstPos**;
} **NSTextCache**;
	This structure describes the current text block and run, and the cursor position in the text.

typedef struct _NSTextChunk {
    short **growby**;
    int **allocated**;
    int **used**;
} **NSTextChunk**;
	NSText uses this structure to implement variable-sized arrays of records.

typedef char *(***NSTextFilterFunc**) (
    id self,
    unsigned char * insertText,
    int *insertLength,
    int position);
	A text filter function implements autoindenting and other features in an NSText object.

typedef int (***NSTextFunc**) (
    id self,
    NSLayInfo *layInfo);
	This is the type for an NSText object's scanning and drawing function, as set through the **setScanFunc:** and **setDrawFunc:** methods.

typedef enum _NSTextAlignment {
    NSLeftTextAlignment,
    NSRightTextAlignment,
    NSCenterTextAlignment,
    NSJustifiedTextAlignment,
    NSNaturalTextAlignment
} **NSTextAlignment**;
	The constants of this type determine text alignment. Used by methods of NSCell, NSControl, NSForm, NSFormCell, and NSText. **NSNaturalTextAlignment** indicates the default alignment for the text.

```
typedef struct _NSTextStyle {
        float indent1st;
        float indent2nd;
        float lineHt;
        float descentLine;
        NSTextAlignment   alignment;
        short   numTabs;
        NSTabStop  *tabs;
} NSTextStyle;
```

NSText uses this structure to describe text layout and tab stops.

```
typedef struct _NSWidthArray {
        NSTextChunk chunk;
        float widths[1];
} NSWidthArray;
```

Holds the character widths for the current line. Since the first field is an NSTextChunk structure you can manipulate the items in the array with the functions that manage variable-sized arrays of records.

```
enum {
        NSLeftTab
};
```

This constant is used by the NSText object's tab functions.

```
enum {
        NSBackspaceKey   = 8,
        NSCarriageReturnKey   = 13,
        NSDeleteKey= 0x7f,
        NSBacktabKey   = 25
};
```

These character-code constants are used by the NSText object's character filter function.

```
enum {
        NSIllegalTextMovement  = 0,
        NSReturnTextMovement  = 0x10,
        NSTabTextMovement   = 0x11,
        NSBacktabTextMovement  = 0x12,
        NSLeftTextMovement   = 0x13,
        NSRightTextMovement   = 0x14,
        NSUpTextMovement   = 0x15,
        NSDownTextMovement   = 0x16
};
```

Movement codes describing types of movement between text fields. Passed in to NSText delegates as the last argument of **textDidEnd:endChar:**.

```
enum {
        NSTextBlockSize   = 512
};
```

The size, in bytes, of a text block.

Break Tables

These tables (with their associated sizes) are finite-state machines that determine word wrapping in an NSText object.

const NSFSM ***NSCBreakTable**;

int **NSCBreakTableSize**;

const NSFSM ***NSEnglishBreakTable**;

int **NSEnglishBreakTableSize**;

const NSFSM ***NSEnglishNoBreakTable**;

int **NSEnglishNoBreakTableSize**;

Character Category Tables

These tables define the character classes used in an NSText object's break and click tables.

const unsigned char ***NSCCharCatTable**;

const unsigned char ***NSEnglishCharCatTable**;

Click Tables

NSText objects use these tables as finite-state machines that determine which characters are selected when the user double-clicks.

const NSFSM ***NSCClickTable**;

int **NSCClickTableSize**;

const NSFSM ***NSEnglishClickTable**;

int **NSEnglishClickTableSize**;

Smart Cut and Paste Tables

These tables are suitable as arguments for the NSText methods **setPreSelSmartable:** and **setPostSelSmartTable:**. When users paste text into an NSText object, if the character to the left (right) side of the new word is not in the left (right) table, an extra space is added to that side.

const unsigned char ***NSCSmartLeftChars**;

const unsigned char ***NSCSmartRightChars**;

const unsigned char ***NSEnglishSmartLeftChars**;

const unsigned char ***NSEnglishSmartRightChars**;

NSCStringText Internal State Structure

This is the structure returned by the **cStringTextInternalState** method of NSCStringText, for use only by applications that need to access the internal state of an NSCStringText object.

typedef struct _NSCStringTextInternalState {

const NSFSM ***breakTable**;	Pointer to state table that specifies word and line breaks
const NSFSM ***clickTable**;	Pointer to state table that defines word boundaries for double-click selection
const unsigned char ***preSelSmartTable**;	Pointer to table that specifies which characters on the left end of a selection are treated as equivalent to a space
const unsigned char ***postSelSmartTable**;	Pointer to table that specifies which characters on the right end of a selection are treated as equivalent to a space
const unsigned char ***charCategoryTable**;	Pointer to table that maps ASCII characters to character classes.
char **delegateMethods**;	Record of notification methods the delegate implements
NSCharFilterFunc **charFilterFunc**;	Function to check each character as it's typed into the text
NSTextFilterFunc **textFilterFunc**;	Function to check text that's being added to the NSCStringText object
NSString ***_string**;	Reserved for internal use
NSTextFunc **scanFunc**;	Function that calculates the line of text
NSTextFunc **drawFunc**;	Function that draws the line of text
id **delegate**;	Object that's notified when the NSCStringText object is modified
int **tag**;	Integer the delegate uses to identify the NSCStringText object
void ***cursorTE**;	Timed entry number for the vertical bar that marks the insertion point
NSTextBlock ***firstTextBlock**;	Pointer to first record in a linked list of text blocks
NSTextBlock ***lastTextBlock**;	Pointer to last record in a linked list of text blocks
NSRunArray ***theRuns**;	Pointer to array of format runs. By default, **theRuns** points to a single run of the default font
NSRun **typingRun**;	Format run to use for the next characters entered
NSBreakArray ***theBreaks**;	Pointer to the array of line breaks
int **growLine**;	Line containing the end of the growing selection
int **textLength**;	Number of characters in the NSCStringText object
float **maxY**;	Bottom of the last line of text, relative to the origin of **bodyRect**
float **maxX**;	Widest line of text. Only accurate after **calcLine** method is invoked
NSRect **bodyRect**;	Rectangle in which the NSCStringText object draws
float **borderWidth**;	Reserved for internal use
char **clickCount**;	Number of clicks that created the selection
NSSelPt **sp0**;	Starting position of the selection
NSSelPt **spN**;	Ending position of the selection
NSSelPt **anchorL**;	Left anchor position
NSSelPt **anchorR**;	Right anchor position
NSSize **maxSize**;	Maximum size of the frame rectangle

NSSize **minSize**;	Minimum size of the frame rectangle
struct **_tFlags** {	
#ifdef __BIG_ENDIAN__	
unsigned int **_editMode**:2;	Reserved for internal use
unsigned int **_selectMode**:2;	Reserved for internal use
unsigned int **_caretState**:2;	Reserved for internal use
unsigned int **changeState**:1;	True if any changes have been made to the text since the NSCStringText object became first responder
unsigned int **charWrap**:1;	True if the NSCStringText object wraps words whose length exceeds the line length on a character basis. False if such words are truncated at end of line
unsigned int **haveDown**:1;	True if the left mouse button (or any button if button functions are not differentiated) is down
unsigned int **anchorIs0**:1;	True if the anchor's position is at **sp0**
unsigned int **horizResizable**:1;	True if the NSCStringText object's width can grow or shrink
unsigned int **vertResizable**:1;	True if the NSCStringText object's height can grow or shrink
unsigned int **overstrikeDiacriticals**:1;	Reserved for internal use
unsigned int **monoFont**:1;	True if the NSCStringText object uses one font for all its text
unsigned int **disableFontPanel**:1;	True if the NSCStringText object doesn't update the font panel automatically
unsigned int **inClipView**:1;	True if the NSCStringText object is a subview of an NSClipView
#else	
unsigned int **inClipView**:1;	
unsigned int **disableFontPanel**:1;	
unsigned int **monoFont**:1;	
unsigned int **overstrikeDiacriticals**:1;	
unsigned int **vertResizable**:1;	
unsigned int **horizResizable**:1;	
unsigned int **anchorIs0**:1;	
unsigned int **haveDown**:1;	
unsigned int **charWrap**:1;	
unsigned int **changeState**:1;	
unsigned int **_caretState**:2;	
unsigned int **_selectMode**:2;	
unsigned int **_editMode**:2;	
#endif	
} **tFlags**;	
void ***_info**;	Reserved for internal use
void ***_textStr**;	Reserved for internal use
} **NSCStringTextInternalState**;	

View

typedef int **NSTrackingRectTag**;

A unique identifier of a tracking rectangle assigned by NSView. (See **addTrackingRectangle:owner: assumeInside:**.)

```
typedef enum _NSBorderType {
        NSNoBorder,
        NSLineBorder,
        NSBezelBorder,
        NSGrooveBorder
} NSBorderType;
```

Constants representing the four types of borders that can appear around NSView objects.

```
enum {
        NSViewNotSizable,
        NSViewMinXMargin,
        NSViewWidthSizable,
        NSViewMaxXMargin,
        NSViewMinYMargin,
        NSViewHeightSizable,
        NSViewMaxYMargin
};
```

NSView uses these autoresize constants to describe the parts of a view (or its margins) that are resized when the view's superview is resized.

Window

```
enum {
        NSNormalWindowLevel   = 0,
        NSFloatingWindowLevel  = 3,
        NSDockWindowLevel   = 5,
        NSSubmenuWindowLevel  = 10,
        NSMainMenuWindowLevel  = 20
};
```

These constants list the window-device tiers that the Application Kit uses. Windows are ordered (or "layered") within tiers: The uppermost window in one tier can still be obscured by the lowest window in the next higher tier.

```
enum {
        NSBorderlessWindowMask,
        NSTitledWindowMask,
        NSClosableWindowMask,
        NSMiniaturizableWindowMask,
        NSResizableWindowMask
};
```

Bitmap masks to determine certain window styles.

Size Globals

These global constants give the dimensions of an icon and contained.

NSSize **NSIconSize**;

NSSize **NSTokenSize**:

Workspace

Workspace File Type Globals

Identifies the type of file queried by the method **getInfoForFile:application:type:** (passed back by reference in last argument).

NSString ***NSPlainFileType**;

NSString ***NSDirectoryFileType**;

NSString ***NSApplicationFileType**;

NSString ***NSFilesystemFileType**;

NSString ***NSShellCommandFileType**;

Workspace File Operation Globals

Used as file-operation arguments in the **performFileOperation:source:destination:files:options:** method (first argument).

NSString ***NSWorkspaceCompressOperation**;

NSString ***NSWorkspaceCopyOperation**;

NSString ***NSWorkspaceDecompressOperation**;

NSString ***NSWorkspaceDecryptOperation**;

NSString ***NSWorkspaceDestroyOperation**;

NSString ***NSWorkspaceDuplicateOperation**;

NSString ***NSWorkspaceEncryptOperation**;

NSString ***NSWorkspaceLinkOperation**;

NSString ***NSWorkspaceMoveOperation**;

NSString ***NSWorkspaceRecycleOperation**;

2 *Foundation Kit*

Introduction

The Foundation Kit defines a base layer of Objective C classes for OpenStep. In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective C language. The Foundation Kit is designed with these goals in mind:

- To provide a set of basic utility classes
- To make software development easier by introducing consistent conventions for things such as deallocation
- To support Unicode strings, object persistence, and object distribution
- To provide a level of operating system independence, enhancing application portability

Classes

The Foundation Kit includes the root class for almost all OpenStep classes, classes representing basic data types such as strings and byte arrays, collections of other objects, and classes representing system information such as dates. The following diagram shows the inheritance relationship among these classes. After the diagram, the specifications for these classes are arranged in alphabetical order.

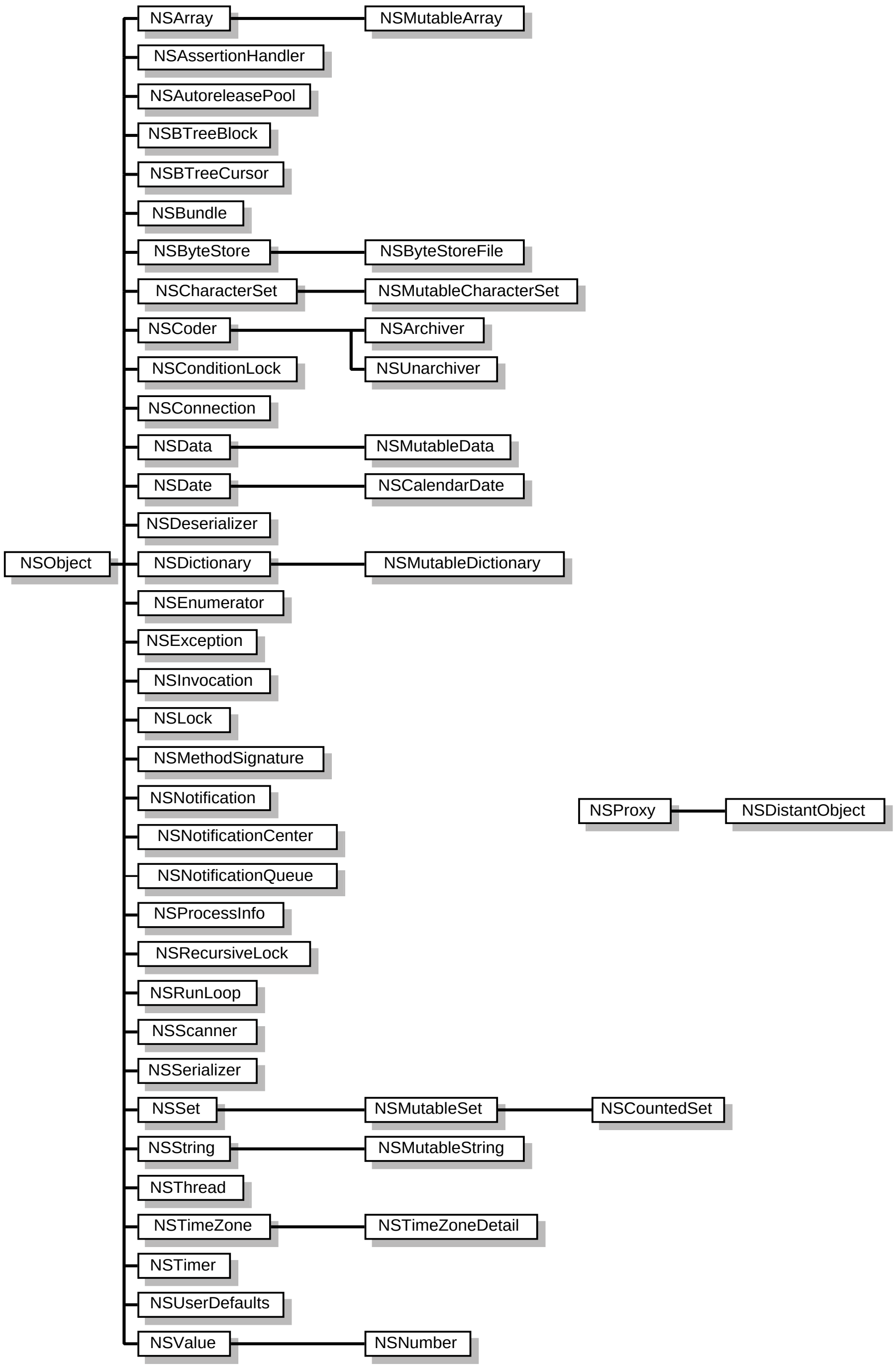

Figure 2-1. Foundation Kit Classes

NSArchiver

Inherits From: NSCoder : NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSArchiver.h

Class Description

NSArchiver, a concrete subclass of NSCoder, defines an object that encodes Objective C objects into an architecture-independent format that can be stored in a file. When objects are archived, their class information and the values of their instance variables are written to the archive. NSArchiver's companion class, NSUnarchiver, takes an archive file and decodes its contents into a set of objects equivalent to the original one.

Archiving is typically initiated by sending an NSArchiver an **encodeRootObject:** or **archiveRootObject:toFile:** message. These messages specify a single object that is the starting point for archiving. The root object receives an **encodeWithCoder:** message (see the NSCoding protocol) that allows it to begin archiving itself and the other objects that it's connected to. An object responds to an **encodeWithCoder:** message by writing its instance variables to the archiver.

An object doesn't have to archive the values of each of its instance variables. Some values may not be important to reestablish and others may be derivable from related state upon unarchiving. Other instance variables should be written to the archive only under certain conditions, as explained below.

NSArchiver overrides the inherited **encodeRootObject:** and **encodeConditionalObject:** methods to support the conditional archiving of members of a graph of objects. When an object receives an **encodeWithCoder:** message, it should respond by unconditionally archiving instance variables that are intrinsic to its nature (with the exceptions noted above) and conditionally archiving those that are not. For example, an NSView unconditionally archives its array of subviews (using **encodeObject:**, or the like) but conditionally archives its superview (using **encodeConditionalObject:**). The archiving system notes each reference to a conditional object, but doesn't actually archive the object unless some other object in the graph requests the object to be archived unconditionally. This ensures that an object is only archived once despite multiple references to it in the object graph. When the objects are extracted from the archive, the multiple references to objects are resolved, and an equivalent graph of objects is reestablished.

Initializing an NSArchiver

– (id)**initForWritingWithMutableData:**(NSMutableData *)*mdata*

Initializes an archiver, encoding stream and version information into mutable data *mdata*. Raises NSInvalidArgumentException if the *mdata* argument is **nil**.

Archiving Data

+ (NSData *)**archivedDataWithRootObject:**(id)*rootObject*

Creates and returns a data object after initializing an archiver with that object and encoding the archiver with *rootObject*.

+ (BOOL)**archiveRootObject:**(id)*rootObject* **toFile:**(NSString *)*path*

Archives *rootObject* by encoding it as a data object in an archiver and writing that data object to file *path*. Returns YES upon success.

– (void)**encodeArrayOfObjCType:**(const char *)*type* **count:**(unsigned int)*count* **at:**(const void *)*array*

Encodes an *array* of *count* data elements of the same Objective C data *type*.

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

Encodes into the linearized data a conditional *object* that points back toward a root object. If **nil** is specified for *object*, it encodes it as **nil** unconditionally. Raises an NSInvalidArgumentException if no root object has been encoded.

– (void)**encodeRootObject:**(id)*rootObject*

Encodes the *rootObject* at the start of the linearized data representing the object graph. Raises an NSInvalidArgumentException if the root object has already been encoded.

Getting Data from the NSArchiver

– (NSMutableData *)**archiverData**

Returns the data object, in mutable form, that is associated with the receiving NSArchiver.

Substituting One Class for Another

– (NSString *)**classNameEncodedForTrueClassName:**(NSString *)*trueName*

Returns the class name used to archive instances of the class *trueName*. See **encodeClassName:intoClassName**.

– (void)**encodeClassName:**(NSString *)*trueName* **intoClassName:**(NSString *)*inArchiveName*)

Encodes in the archive a substitute class name for the real class name (*trueName*).

NSArray

Inherits From: NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying
NSObject (NSObject)

Declared In: Foundation/NSArray.h

Class Description

The NSArray class declares the programmatic interface to an object that manages an immutable array of objects. (The complementary class NSMutableArray manages modifiable arrays of objects.) NSArray's two primitive methods—**count** and **objectAtIndex:**—provide the basis for all the other methods in its interface. The **count** method returns the number of elements in the array. **objectAtIndex:** gives you access to the array elements by index, with index values starting at 0.

The methods **objectEnumerator** and **reverseObjectEnumerator** also permit sequential access of the elements of the array, differing only in the direction of travel through the elements. These methods are provided so that array objects can be traversed in a manner similar to that used for objects of other collection classes, such as NSDictionary.

Generally, you instantiate an NSArray by sending one of the **array...** messages to the NSArray class object. These methods return an NSArray containing the elements you pass in as arguments. (Note that arrays can't contain **nil** objects.) These objects aren't copied; rather, each object receives a **retain** message before it's added to the array. When an object is removed from an array, it's sent a **release** message.

NSArray provides methods for querying the elements of the array. **indexOfObject:** searches the array for the object that matches its argument. To determine whether the search is successful, each element of the array is sent an **isEqual:** message, as declared in the NSObject protocol. Another method, **indexOfObjectIdenticalTo:**, is provided for the less common case of determining whether a specific object is present in the array. **indexOfObjectIdenticalTo:** tests each element in the array to see whether its **id** matches that of the argument.

NSArray's **makeObjectsPerform:** and **makeObjectsPerform:withObject:** methods let you act on the individual objects in the array by sending them messages. To act on the array as a whole, a variety of methods are defined. You can create a sorted version of the array (**sortedArrayUsingSelector:** and **sortedArrayUsingFunction:context:**), extract a subset of the array (**subarrayWithRange:**), or concatenate the elements of an array of NSString objects into a single string (**componentsJoinedByString:**). In addition, you can compare two array objects using the **isEqualToArray:** and **firstObjectCommonWithArray:** methods.

Allocating and Initializing an Array

+ (id)**allocWithZone:**(NSZone *)*zone* — Returns an uninitialized array object in *zone*.

+ (id)**array** — Returns an empty array object

+ (id)**arrayWithObject:**(id)*anObject* — Returns an NSArray containing the single element *anObject.* Raises an NSInvalidArgumentException if *anObject* is **nil**.

+ (id)**arrayWithObjects:**(id)*firstObj,...* — Returns an NSArray containing the objects in the argument list. The object list is comma-separated and ends with **nil**.

– (NSArray *)**arrayByAddingObject:**(id)*anObject* — Returns an NSArray containing the receiver's elements plus *anObject*.

– (NSArray *)**arrayByAddingObjectsFromArray:**(NSArray *)*anotherArray* — Returns an NSArray containing the receiver's elements plus the elements from *anotherArray.*

– (id)**initWithArray:**(NSArray *)*anotherArray* — Initializes a newly allocated array object by placing in it the objects contained in *anotherArray.*

– (id)**initWithObjects:**(id)*firstObj,...* — Initializes a newly allocated array object by placing in it the objects in the argument list. The object list is comma-separated and ends with **nil**. Raises an NSInvalidArgumentException if any object in the list of objects is **nil**.

– (id)**initWithObjects:**(id *)*objects* **count:**(unsigned int)*count* — Initializes a newly allocated array object by placing in it *count* objects from the *objects* array

Querying the Array

– (BOOL)**containsObject:**(id)*anObject* — Returns YES if *anObject* is present in the array.

– (unsigned int)**count** — Returns the number of objects currently in the array.

– (unsigned int)**indexOfObject:**(id)*anObject* — Returns the index of *anObject*, if found; otherwise, returns NSNotFound. This method checks the elements in the array from last to first by sending them an **isEqual:** message.

– (unsigned int)**indexOfObjectIdenticalTo:**(id)*anObject* — Returns the index of *anObject*, if found; otherwise, returns NSNotFound. This method checks the elements in the array from last to first by comparing their **id**s.

– (id)**lastObject** — Returns the last object in the array.

– (id)**objectAtIndex:**(unsigned int)*index*

Returns the object located at *index*. An array's index starts at 0. Raises an NSRangeException if *index* is beyond the end of the array.

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that lets you access each object in the array, starting with the first element.

– (NSEnumerator *)**reverseObjectEnumerator**

Returns an enumerator object that lets you access each object in the array, from the last element to the first.

Sending Messages to Elements

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

Sends an *aSelector* message to each object in the array.

– (void)**makeObjectsPerform:**(SEL)*aSelector* **withObject:**(id)*anObject*

Sends an *aSelector* message to each object in the array, with *anObject* as an argument.

Comparing Arrays

– (id)**firstObjectCommonWithArray:**(NSArray *)*otherArray*

Returns the first object from the receiver's array that's equal to an object in *otherArray.*

– (BOOL)**isEqualToArray:**(NSArray *)*otherArray*

Compares the receiving array object to *otherArray.*

Deriving New Arrays

– (NSArray *)**sortedArrayUsingFunction:**(int(*)(id *element1*, id *element2,* void **userData*))*comparator* **context:**(void *)*context*

Returns an array listing the receiver's elements in ascending order as defined by the comparison function *comparator. context* is passed to the comparator function as its third argument.

– (NSArray *)**sortedArrayUsingSelector:**(SEL)*comparator*

Returns an array listing the receiver's elements in ascending order, as determined by the comparison method specified by the selector *comparator.*

– (NSArray *)**subarrayWithRange:**(NSRange)*range*

Returns an array containing the receiver's elements that fall within the limits specified by *range*.

Joining String Elements

– (NSString *)**componentsJoinedByString:**(NSString *)*separator*

Returns a string that's the result of interposing *separator* between the elements of the receiver's array.

Creating a String Description of the Array

– (NSString *)**description**

Returns a string object that represents the contents of the receiver.

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary*

Returns a string representation of the NSArray object. Included are the key and values that represent the locale data from *localeDictionary.*

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary* **indent:**(unsigned int)*level*

Returns a string representation of the NSArray object. Included are the key and values that represent the locale data from *localeDictionary.* Elements of the array are indented from the left margin by *level* + 1 multiples of four spaces, to make the output more readable.

NSAssertionHandler

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSExceptions.h

Class Description

An *assertion* is a statement about conditions during the execution of program code, such as the relationship between variables, the state of a boolean variable, the value of an expression, and so on. If the statement about the conditions proves false, the assertion is said to have failed, and usually some action must be taken to report the failed assertion. Application programmers wishing to provide more detailed control over assertion failures than provided by the macros defined below would use the methods of NSAssertionHandler to report assertion failures.

NSAssertionHandler provides a mechanism whereby each distinct thread of execution can have a separate handler to deal with failed assertions in code. The *fileName* and *line* arguments to the methods described below can be obtained by using the **`__FILE__`** and **`__LINE__`** macros that are pre-defined in the C pre-processor.

The **Foundation/NSExceptions.h** header file contains a collection of macros that can be used to state assertions within methods, and contains a parallel collection of macros that can be used to state assertions within regular C functions. If the condition tested in any of these macros fails, the current assertion handler is invoked with one of the methods defined below, depending on whether the macro is one of the NSAssertN or one of the NSCAssertN macros. Separate macros have from 1 to 5 arguments. Macros for dealing with assertion failures within methods are:

```
NSAssert1(condition, description, argument1);
NSAssert2(condition, description, argument1, argument2);
NSAssert3(condition, description, argument1, argument2, argument3);
NSAssert4(condition, description, argument1, argument2, argument3, argument4);
NSAssert5(condition, description, argument1, argument2, argument3, argument4, argument5);
```

In each case, *condition* is the statement to be tested (for example, **index < length**), *description* is a description of the reason for the failure (in the form of a printf-style format NSString), and each *arg*N is an argument to be formatted according to the *description* string.

The parallel set of macros for dealing with failed assertions from within C functions have names of the form NSCAssertN instead of NSAssertN. The arguments are otherwise the same as the NSAssertN macros.

Getting the Current Handler

+ (NSAssertionHandler *)**currentHandler**	Returns the assertion handler for the current thread.

Handling Failures

– (void)**handleFailureInFunction:**(NSString *)*functionName*
file:(NSString *)*fileName*
lineNumber:(int)*line*
description:(NSString *)*format,...*

Logs an error message that includes *functionName*; the source file *fileName* and the *line* number where the failure occured; and a short description of the failure, described by *format*. It then raises an NSInternalInconsistencyException.

– (void)**handleFailureInMethod:**(SEL)*selector*
object:(id)*object*
file:(NSString *)*fileName*
lineNumber:(int)*line*
description:(NSString *)*format,...*

Logs an error message that includes the method (*selector*) and *object* associated with the failure; the source file *fileName* and *line* number in that file where the failure occured; and a short description of the failure, described by *format*. It then raises an NSInternalInconsistencyException.

NSAutoreleasePool

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSAutoreleasePool.h

Class Description

The Foundation Kit uses the NSAutoreleasePool class to implement NSObject's **autorelease** method. An autorelease pool simply contains other objects, and when deallocated, sends a **release** message to each of those objects. An object can be put into the same pool several times, and receives a **release** message for each time it was put into the pool.

You use autorelease pools to limit the time an object remains valid after it's been "autoreleased" (that is, after it's been sent an **autorelease** message or has otherwise been added to an autorelease pool). Autorelease pools are created using the usual **alloc** and **init** messages, and disposed of with **release**. An autorelease pool should always be released in the same context (invocation of a method or function, or body of a loop) that it was created. You should never send **retain** or **autorelease** messages to an autorelease pool.

Autorelease pools are automatically created and destroyed in OpenStep applications, so your code normally doesn't have to worry about them. There are two cases, though, where you should explicitly create and destroy your own autorelease pools. If you're writing a program that's not based on the Application Kit, such as a UNIX tool, there's no built-in support for autorelease pools; you must create and destroy them yourself. Also, if you need to write a loop that creates many temporary objects, you should create an autorelease pool in the loop to prevent too long a delay in the disposal of those objects.

Enabling the autorelease feature in a program that's not based on the Application Kit is easy. Many programs have a top-level loop where they do most of their work. To enable the autorelease feature you create an autorelease pool at the beginning of this loop and release it at the end. An **autorelease** message sent in the body of the loop automatically puts its receiver into this pool. The **main()** function might look like this:

```
int main(int argc, char *argv[])
{
    int i;

    /* Do whatever setup is needed. */
    for (i = 0; i < argc; i++) {
        NSAutoreleasePool *pool;
        NSString *fileContents;

        NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
        fileContents = [[[NSString alloc] initWithContentsOfFile:argv[i]] autorelease];
        processFile(fileContents);
        [pool release];
    }

    /* Do whatever cleanup is needed. */
    exit(EXIT_SUCCESS);
}
```

Any object autoreleased inside the **for** loop, such as the **fileContents** string object, is added to **pool**, and when **pool** is released at the end of the loop those objects are also released.

Note that autoreleasing doesn't work outside of the loop. This isn't a problem, since the program terminates shortly after the loop ends, and memory leaks aren't usually serious at that stage of execution. Your cleanup code shouldn't refer to any objects created inside the loop, though, since they may be autoreleased in the loop and therefore released as soon as it ends.

Nesting Autorelease Pools

You may need to manually create and destroy autorelease pools even in an application that uses the Application Kit if you write loops that create many temporary objects. For example, if you write a loop that iterates 1000 times and invokes a method that creates 15 temporary objects, those 15,000 objects will remain until the application's autorelease pool is deallocated, possibly well after they're no longer needed.

You can create your own autorelease pools within the loop to prevent these unwanted objects from remaining around. Autorelease pools nest themselves on a per-thread basis, so that if you create your own pool, it adds itself to the application's default pool, forming a stack of autorelease pools. Likewise, if you create another pool (within a nested loop, perhaps), it adds itself to the first pool you created. **autorelease** automatically adds its receiver to the last pool created, creating a nesting of autorelease contexts. The implications of this are described below.

A method that creates autorelease pools looks much like the **main()** function given above:

```
- (void)processString:(NSString *)aString
{
    int i;

    for (i = 0; i < 1000; i++) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];
        NSString *thisLine;

        thisLine = [self lineNumbered:i fromString:aString];
        /* Do some work with thisLine. */
        [subpool release];
    }
    return;
}
```

If you assume that **lineNumbered:fromString:** returns a string object that's been autoreleased while **subpool** is in effect, that object is released with **subpool** at the end of the loop. The work involving **thisLine** may create other temporary objects, which are also released at the end of the loop. None of these objects remains outside of this loop or the **processString:** method (unless they've been retained).

Note that because an autorelease pool adds itself to the previous pool when created, it doesn't cause a memory leak in the face of an exception or other sudden transfer out of the current context. If an exception occurs in the above loop, or if the work in the loop involves immediately returning or breaking out of the loop, the sub-pool is released by the application's default pool (or whatever pool was in effect before the sub-pool was created), "unwinding" the autorelease-pool stack up to the one that's supposed to be active.

Guaranteeing the Foundation Ownership Policy

By manually creating an autorelease pool, you reduce the potential lifetime of temporary objects to the lifetime of that pool. After an autorelease pool is deallocated, you should regard as "disposed of" any object that was autoreleased while that pool was in effect, and not send a message to that object or return it to the invoker of your method. This method, for example, is incorrect:

```
– findMatchingObject:anObject
{
    id match = nil;
    while (match == nil) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];

       /* Do some searching that creates a lot of temporary objects.*/

        match = [self expensiveSearchForObject:anObject];
        [subpool release];
    }
    /* Danger!! The match object may not exist at this point! */
    [match setIsMatch:YES forObject:anObject];
    return match;
}
```

expensiveSearchForObject: is invoked while **subpool** is in effect, which means that **match**, which may have been autoreleased, is released at the bottom of the loop. Sending **setIsMatch:forObject:** after the loop could cause the application to crash. Similarly, returning **match** allows the sender of **findMatchingObject:** to send a message to it, also causing your application to crash.

If you must pull a temporary object out of a nested autorelease context, you can do so by retaining the object within the context and then autoreleasing it after the pool has been released. Here's a correct implementation of **findMatchingObject:**:

```
– findMatchingObject:anObject
{
    id match = nil;
    while (match == nil) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];



        /* Do a search that creates a lot of temporary objects. */

        match = [self expensiveSearchForObject:anObject];
        if (match != nil) [match retain]; /* Keep match around. */
        [subpool release];
    }
    [match setIsMatch:YES forObject:anObject];
    return [match autorelease];   /* Let match go and return it. */
}
```

By retaining **match** while **subpool** is in effect and autoreleasing it after the **subpool** has been released, **match** is effectively moved from **subpool** to the pool that was previously in effect. This gives it a longer lifetime and allows it to be sent messages outside the loop and to be returned to the invoker of **findMatchingObject:**.

General Exception Conditions

An NSInvalidArgumentException is raised on any attempt to send either **retain** or **autorelease** messages to an autorelease pool object.

Adding an Object to the Current Pool

+ (void)**addObject:**(id)*anObject* — Adds *anObject* to the active autorelease pool in the current thread.

Adding an Object to a Pool

– (void)**addObject:**(id)*anObject* — Adds *anObject* to the receiver.

NSBTreeBlock

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSByteStore.h

Class Description

An NSBTreeBlock provides ordered, associative storage and retrieval of untyped data. It identifies and orders data items, called *values*, by *key*, using a comparator function. A companion class, NSBTreeCursor, actually manipulates the contents of the NSBTreeBlock; NSBTreeBlock only provides the mechanisms for storing and sorting the key/value pairs.

Setting Up an NSBTreeBlock

An NSBTreeBlock can be used with either a memory-based NSByteStore or an NSByteStoreFile. The NSByteStore holds the contents of the NSBTreeBlock. Use NSBTreeBlock with NSByteStoreFile to build persistent databases. An NSBTreeBlock is initialized as a new client of an NSByteStore using the method **initWithStore:** or **initWithStore:block:**. The NSBTreeBlock takes up one block in the NSByteStore per key/value pair and one block for each node in the tree. An NSBTreeBlock will always take up at least one block in the NSByteStore.

After the NSBTreeBlock has been initialized, it must have its comparator function set with the **setComparator:context:**. A comparator function takes as arguments two pieces of arbitrary data and their lengths and returns an integer indicating their ordering relative to one another. A comparator function is of type (NSBTreeComparator *), which has the form:

```
typedef int NSBTreeComparator(NSData * data1, NSData * data2, const void *context)
```

where *data1* and *data2* are pointers to data and *context* is a pointer to blind data that may be used by the comparator function. The comparator function returns a number less than 0 if *data1* is considered less than *data2*, greater than 0 if *data1* is considered greater than *data2*, and equal to 0 if *data1* and *data2* are considered equal. By default, NSBTreeBlocks compare keys as strings.

Getting Data Into and Out of an NSBTreeBlock

As stated above, NSBTreeBlock simply provides the capacity for associative storage. An NSBTreeCursor is needed to take advantage of that capacity. An NSBTreeCursor is like a pointer into the NSBTreeBlock: It can move to specific positions within the key space and perform operations on the values stored at those locations, independent of other cursors. See the NSBTreeCursor class description for more information.

Multiple NSBTreeCursors may independently access a single NSBTreeBlock. The actions of one cursor don't affect any of the other cursors in the NSBTreeBlock, except to the extent that they modify the contents of the NSBTreeBlock. It is both safe and meaningful to remove a record that another NSBTreeCursor has just located, as long as the code using the other NSBTreeCursor anticipates this possibility, as described below.

In the case of one cursor removing a value that another cursor has just located, the second cursor will have received an indication from a key-locating method (for example, **moveCursorToKey:**) that it has found a key. When it tries to access the value associated with that key, however, the key may no longer exist. The cursor will detect the deletion and slide forward to the next available key if asked to read the value, or it will raise an exception if asked to remove or write the value. If your code allows multiple cursors to be concurrently active in a single NSBTreeBlock, it must anticipate this behavior by handling the exceptions that may be raised and by comparing the key against the expected value after invoking **cursorKey**. If one cursor is pointed at a key and a second cursor removes or adds a key at a different location, it does not change the position of the first cursor.

Working With the NSByteStore

Since NSBTreeBlock is an NSByteStore client, the transaction model of NSByteStore applies to changes made to the contents of an NSBTreeBlock. In particular, you must send the **commitTransaction** message to the NSByteStore to have changes to the NSBTreeBlock take effect (and be flushed to disk for a file-based store). If an NSBTreeBlock is used on a strictly read-only basis, transaction management can be ignored.

After an **abortTransaction**, a cursor may be pointing to a key that no longer exists. Therefore, you must reposition each cursor using one of the **moveCursor...** methods after an **abortTransaction**.

Creating and Initializing a New NSBTreeBlock Instance

+ (NSBTreeBlock *)**btreeBlockWithStore:**(NSByteStore *)*aStore*

Returns a new NSBTreeBlock instance in the designated NSByteStore.

+ (NSBTreeBlock *)**btreeBlockWithStore:**(NSByteStore *)*aStore* **block:**(unsigned)*aBlock*

Returns a new NSBTreeBlock instance in the designated NSByteStore with *aBlock* as the root block of the NSBTreeBlock. If *aBlock* does not exist or is invalid, the NSBTreeInitException is raised.

– (id)**initWithStore:**(NSByteStore *)*aStore*

Initializes a newly allocated NSBTreeBlock instance in the designated NSByteStore.

– (id)**initWithStore:**(NSByteStore *)*aStore* **block:**(unsigned)*aBlock*

Initializes a newly allocated NSBTreeBlock instance in the designated NSByteStore with *aBlock* as the root block of the NSBTreeBlock. If *aBlock* does not exist or is invalid, the NSBTreeInitException is raised.

Accessing Information About the NSByteStore

– (NSByteStore *)**byteStore**

Returns the NSByteStore associated with the NSBTreeBlock.

– (unsigned)**storeBlock**

Returns the number of the NSByteStore block that contains the root of the NSBTreeBlock.

Setting the Comparator

– (void)**setComparator:**(NSBTreeComparator *)*comparator*
context:(const void *)*context*

Sets the comparison method. The default is string comparison. When a value is inserted in the NSBTreeBlock, the comparator function decides where to put it. For more information, see the class description.

Accessing NSBTreeBlock information

– (unsigned)**count**

Returns the number of key/value pairs stored in the NSBTreeBlock.

Affecting NSBTreeBlock Contents

– (void)**removeAllObjects**

Removes all key/value pairs from the NSBTreeBlock.

NSBTreeCursor

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSByteStore.h

Class Description

An NSBTreeCursor provides access to the keys and values stored in an NSBTreeBlock. It's essentially a pointer into the NSBTreeBlock's key space, and may be positioned by key to perform operations on the value stored at a given location.

An NSBTreeCursor works with a single NSBTreeBlock, but several NSBTreeCursors may access the same NSBTreeBlock and be positioned independently without conflict. See the NSBTreeBlock class specification for more information on concurrent access with multiple NSBTreeCursors.

Positioning the Cursor and Accessing Data

NSBTreeCursor contains methods that walk through the key/value pairs in the NSBTreeBlock. The method **moveCursorToFirstKey** will point the cursor to the first key in the key space, and you can use **moveCursorToNextKey** to essentially walk through all of the keys in the NSBTreeBlock. To point the cursor at a specific key/value pair, use **moveCursorToKey:**. This method returns YES if it finds the key and NO if it does not. If **moveCursorToKey:** returns NO, it still points the cursor at that key. For example, suppose the keys into the key space are integer IDs divisible by 10, and you call **moveCursorToKey:** with 54 as the key. (In reality, keys must be NSData objects, but to make this example more clear, it uses integers.) There is no key 54, so **moveCursorToKey:** returns NO, but the cursor points to where key 54 would be if it existed. A subsequent call to **moveCursorToNextKey** would point the cursor at key 60. The method **isOnKey** tells you if the cursor is pointing to a valid key.

To insert a key/value pair into the NSBTreeBlock, you take advantage of the **moveCursorToKey:** method's return value. Send **moveCursorToKey:** with the key you want to insert. If if returns NO, send **writeValue:** with the value you want to insert. The key/value pair will be inserted.

A cursor at a position with no key can't access a value there. If the cursor is asked to access a value anyway, it has two options: try to find a value or indicate that it can't access one. Where it makes sense, a cursor should try to find a value by sliding forward in the key space to the next actual key. When this isn't possible or desirable, the cursor should indicate that it can't find or access a value, by raising the NSBTreeNoValueException exception. In the previous example, if the cursor is asked to retrieve the information at key 54, the cursor will slide forward and return the information at key 60. At this point, you can use the **cursorKey** method to find out which key the cursor is pointing to. **cursorKey** will return 60 to let you know that the cursor has slid forward.

A cursor cannot write inside (with the method **writeValue:range:**) or remove the value (with the method **removeValue**) at a location where there is no key. Since there is no value, and since writing into part of a value or removing it would change data that the programmer probably doesn't want altered (namely, the value for the next actual key), the NSBTreeCursor will indicate that there is no value to write into by raising the NSBTreeNoValueException exception.

Working With the NSByteStore

Since NSBTreeBlock is an NSByteStore client, the transaction model of NSByteStore applies to changes made to the contents of an NSBTreeBlock. In particular, you must send the **commitTransaction** message to the NSByteStore to have changes to the NSBTreeBlock take effect (and be flushed to disk for a file-based store). If an NSBTreeBlock is used on a strictly read-only basis, transaction management can be ignored.

After an **abortTransaction**, a cursor may be pointing to a key that no longer exists. Therefore, you must reposition each cursor using one of the **moveCursor...** methods after an **abortTransaction**.

Creating and Initializing a New NSBTreeCursor Instance

+ (NSBTreeCursor *)**bTreeCursorWithBTree:**(NSBTreeBlock *)*aBTree*
Returns a new NSBTreeCursor instance and associates it with the *aBTree* object.

– (id)**initWithBTree:**(NSBTreeBlock *)*aBTree* — Initializes a newly allocated NSBTreeCursor instance and associates it with the *aBTree* object.

Obtaining Information about the NSBTreeBlock

– (NSBTreeBlock *)**btree** — Returns the NSBTreeBlock with which the NSBTreeCursor is associated.

Positioning the Cursor

– (BOOL)**moveCursorToFirstKey** — Positions the cursor at the first key in the key space.

– (BOOL)**moveCursorToLastKey** — Positions the cursor at the last key in the key space.

– (BOOL)**moveCursorToNextKey** — Positions the cursor at the next key in the key space. If the cursor is at the last key, it does not move.

– (BOOL)**moveCursorToPreviousKey** — Positions the cursor at the previous key in the key space. If the cursor is at the first key, it does not move.

– (BOOL)**moveCursorToKey:**(NSData *)*key* — Positions the cursor at *key.*

–(BOOL)**isOnKey** — Returns YES if the cursor matched a key on the last operation.

Accessing the Data

– (NSData *)**cursorKey**

Returns the key that the cursor is pointing to.

– (NSData *)**cursorValue**

Returns the value associated with the key that the cursor is pointing to.

– (NSData *)**cursorValueWithRange:**(NSRange)*range*

Returns a portion, specified by *range*, of the value associated with the key that the cursor is pointing to.

Changing the Data in the NSBTreeBlock

– (BOOL)**writeValue:**(NSData *)*value*

Replaces the value associated with the key that the cursor is pointing to, if the key exists. If the key does not exist, it creates a new key/value pair using the key that the cursor is currently pointing to and *value* as the value. This method returns YES if it inserted a new key/value pair and NO if it overwrote an existing value.

– (void)**writeValue:**(NSData *)*value* **atIndex:**(unsigned)*index*

Replaces a portion, starting at *index*, of the value associated with the key that the cursor is pointing to. If the key does not exist, the NSBTreeNoValueException exception is raised.

– (void)**removeValue**

Deletes the key/value pair from the NSBTreeBlock. If the key/value pair already does not exist, the NSBTreeNoValueException exception is raised.

NSBundle

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSBundle.h

Class Description

A *bundle* is a mechanism for grouping application *resources* into convenient chunks. A typical (but by no means the only) application of a bundle is to group executable code together with the resources used by that executable code. A major use of bundles is to handle localization issues, as described below in “Localized Resources”.

An NSBundle is an object that corresponds to a directory (or folder in the terminology of some operating systems) where application resources are stored. The directory, in essence, “bundles” a set of resources used by an application, and the NSBundle object makes those resources available to the application. NSBundle is able to find requested resources in the directory and, in some cases, dynamically load executable code. The term “bundle” is used both for the object and for the directory it represents.

Bundled resources might include such things as:

- Images—TIFF or EPS (for instance) images used by an application’s user interface components
- Sounds
- Localized character strings
- Executable code
- User Interface resources—files describing the layout of user interface objects and their relationships with other objects

Each resource within a bundle usually resides in a separate file.

Localized Resources

If an application is to be used in more than one part of the world, its resources may need to be customized, or “localized”, for language, country, or cultural region. An application may need, for example, to have separate Japanese, English, French, Hindi, and Swedish versions of the character strings that label menu commands.

Resource files specific to a particular language are grouped together in a subdirectory of the bundle directory. The subdirectory has the name of the language (in English) followed by a “.lproj” extension (for “language project”). The application mentioned above, for example, would have **Japanese.lproj**, **English.lproj**, **French.lproj**, **Hindi.lproj**, and **Swedish.lproj** subdirectories.

Each “.lproj” subdirectory in a bundle has the same set of files; all versions of a resource file must have the same name.

The Main Bundle

Every application is considered to have at least one bundle—its *main bundle*—the directory where its executable file is located. If the application is organized into a file package marked by a “.app” extension, the file package is the main bundle.

Other Bundles

An application can be organized into any number of other bundles in addition to the main bundle. For example, an application for managing PostScript printers may have a bundle full of PostScript code to be downloaded to printers.These other bundles usually reside inside the application file package, but they can be located anywhere in the file system. Each bundle directory is represented in the application by a separate NSBundle object.

By convention, bundle directories other than the main bundle end in a “.bundle” extension.

Dynamically Loadable Classes

Any bundle directory can contain a file with executable code. For the main bundle, that file is the application executable that's loaded into memory when the application is launched. The executable in the main bundle includes the **main()** function and other code necessary to start up the application.

Executable files in other bundle directories hold class (and category) definitions that the Bundle object can dynamically load while the application runs. When asked, the Bundle returns class objects for the classes (and categories) stored in the file. It waits to load the file until those classes are needed.

By using a number of separate bundles, you can split an application into smaller, more manageable pieces. Each piece is loaded into memory only when the code being executed requires it, so the application can start up faster than it otherwise would. And, assuming users will rarely exercise every part of an application, the application will also consume less memory as it runs.

The file that contains dynamically loadable code must have the same name as the bundle directory, but without the “.bundle” extension.

Since each bundle can have only one executable file, that file should be kept free of localizable content. Anything that needs to be localized should be segregated into separate resource files and stored in “.lproj” subdirectories.

Working with Bundles

Generally, you instantiate a bundle object by sending one of the **bundleForClass:**, **bundleWithPath:**, or **mainBundle** methods to the NSBundle class object. **mainBundle** gives you the NSBundle object corresponding to the directory containing the application's executable.

Initializing an NSBundle

– (id)**initWithPath:**(NSString *)*path*	Initializes a newly allocated NSBundle object to make it the NSBundle for the *path* directory.

Getting an NSBundle

+ (NSBundle *)**bundleForClass:**(Class)*aClass*	Returns the NSBundle object that dynamically loaded *aClass*, or the main bundle object if *aClass* wasn't dynamically loaded.
+ (NSBundle *)**bundleWithPath:**(NSString *)*path*	Returns an NSBundle object that's initialized for the *path* directory.
+ (NSBundle *)**mainBundle**	Returns the NSBundle object that corresponds to the directory where the application executable is located.

Getting a Bundled Class

– (Class)**classNamed:**(NSString *)*className*	Returns the class object for the *className* class, or **nil** if *className* isn't one of the classes associated with the receiver.
– (Class)**principalClass**	Returns the class object for the *first* class that's dynamically loaded by the NSBundle, or **nil** if the NSBundle can't dynamically load any classes.

Finding a Resource

+ (NSString *)**pathForResource:**(NSString *)*name* **ofType:**(NSString *)*ext* **inDirectory:**(NSString *)*bundlePath* **withVersion:**(int)*version*	Returns the path for the resource identified by *name*, having the specified filename *ext*, residing in *bundlePath*, and having version number *version*.
– (NSString *)**pathForResource:**(NSString *)*name* **ofType:**(NSString *)*ext*	Returns the path for the resource identified by *name* having the specified filename extension *ext*.

Getting the Bundle Directory

– (NSString *)**bundlePath**

Returns a string containing the full pathname of the receiver's bundle directory.

Stripping Symbols

+ (void)**stripAfterLoading:**(BOOL)*flag*

Sets whether symbols are stripped when modules are loaded. The default is YES. You would usually set *flag* to NO for debugging purposes.

Managing Localized Resources

– (NSString *)**localizedStringForKey:**(NSString *)*key*
value:(NSString *)*value*
table:(NSString *)*tableName*

Returns a localized version of the string designated by *key*. *tableName* specifies the string table to search; if *tableName* is NULL, the file **Localizable.strings** is used. *value* specifies the value to return if the key or table can't be found (or if *key* is NULL).

Setting the Version

– (unsigned)**bundleVersion**

Returns the version last set by the **setBundleVersion:** method, or 0 if no version has been set.

– (void)**setBundleVersion:**(unsigned)*version*

Sets the version that the NSBundle will use when searching ".lproj" subdirectories for resource files.

NSByteStore

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSByteStore.h

Class Description

An NSByteStore object manages a single memory-based heap. Use NSByteStore to allocate storage in data-intensive applications. Its main feature is transaction management, which makes compound operations atomic and ensures data integrity.

You address the blocks of storage that an NSByteStore manages through unsigned integers called block numbers. To gain access to the contents of a block, you first must open the block for reading or writing. When you open a block, the NSByteStore resolves the block number into a pointer. While a block is open, you can address its contents using the pointer and can safely assume that the block won't move. Once you close the block, however, the NSByteStore is free to move it in order to compact storage; so the pointer may become invalid.

The contents of an NSByteStore are relocatable to and from other instances of NSByteStore and its subclasses. Although the address of a block becomes invalid when the block is relocated, its block number remains constant. Since block numbers are indirect references to data, it's possible to retrieve the contents of an NSByteStore without invalidating block number-based referential data structures residing in the NSByteStore, like linked lists or trees. This makes it easy to copy complex structures or to quickly save them to a file.

A subclass of NSByteStore, NSByteStoreFile, stores data in a file so that you can retain data created and changed by your application. For more information, see its class description.

Transactions

NSByteStore implements *transactions*, allowing several operations to be grouped together in such a way that either all of them take effect, or none of them do. Transactions help to ensure semantic integrity by making compound operations atomic, and they provide a convenient way to undo a series of changes. If you use NSByteStoreFile, the use of transactions also ensures data integrity against process and system crashes. This means that if a system loses power, the NSByteStoreFile's contents can be recovered intact on power up, in the state they were in after the last transaction that actually finished.

Transactions are either enabled or disabled for an object. Most likely, you will want to disable transactions for NSByteStores (unless you want the undo capability) and enable them for NSByteStoreFiles. When transactions are enabled, NSByteStore copies blocks that your application opens for writing. Thus, updates are slower when transactions are enabled. If you are using NSByteStore directly, its contents are always destroyed by a system crash, so the only advantage to using transactions is the undo capability. If you are using NSByteStoreFile, enabling transactions may save some of the changes made before a system crash. Therefore, you should always use transactions with NSByteStoreFile except if it contains data that can be easily reconstructed, such as an index.

Using Transactions

A single transaction begins with a **startTransaction** message and ends with either a **commitTransaction** or **abortTransaction** message. **startTransaction** enables transactions if they are disabled. Sending **commitTransaction** means you want the changes made by this transaction to take effect. **abortTransaction** means you want to cancel the changes made by this transaction.

You can check whether transactions have been enabled with **areTransactionsEnabled**. You may want to do this if your code is invoked by higher level methods that determine the transaction management policy for the application. For example, NSByteStore uses **areTransactionsEnabled** to determine whether or not to invoke **startTransaction** before responding to an **empty** message.

You can nest transactions. The first **startTransaction** message (or the first message that opens a block after **enableTransactions**) starts transaction 1. If you send **startTransaction** again before ending transaction 1, it begins transaction 2, which is nested inside transaction 1. The **nestingLevel** method returns the current nesting level of transactions. **startTransaction** also returns the nesting level as the transaction's ID.

The trick with nesting transactions is: *the changes a transaction makes aren't really made until the nesting level returns to 0*. In other words, changes don't actually take effect until the top-level transaction is committed. This means that any blocks that any of the transactions have opened for writing will not be available until the all of the transactions are finished. So, if you start a transaction at nesting level 2, make some changes to blocks 3, 5, and 7, and then you send **commitTransaction**, all that **commitTransaction** really does is set the nesting level to 1 and tell transaction 1 about the changes to blocks 3, 5, and 7. If you then send **commitTransaction** at transaction 1, **commitTransaction** sets the nesting level to 0. Because the nesting level is now 0, the changes can take place. Blocks 3, 5, and 7 are overwritten with the changes made during transaction 2 and are made available. If instead you decide to abort transaction 1 (by sending **abortTransaction**), the changes transaction 2 made to blocks 3, 5, and 7 are cancelled, as well as any changes transaction 1 made to any blocks. In this way, the parent of a transaction can undo changes made by their children, but the children cannot undo the changes made by their parents.

Note that if your code makes changes outside any transaction while transactions are enabled, an enclosing transaction is started automatically. The next invocation of **startTransaction**, if any, before an intervening abort or commit, simply picks up this enclosing transaction and reports a nesting level of 1. Thus, if nesting isn't needed, your code can simply enable transactions initially with a pair of **startTransaction**/**commitTransaction** messages, and thereafter use only **commitTransaction** to mark transaction boundaries. New transactions implicitly begin with the first modification following each commit.

Any modifications that haven't been committed are aborted when an NSByteStore is freed.

Opening Blocks for Reading or Writing

When you open a block for reading or writing, that block is unavailable until you specify that you are finished.

When you are finished reading a block, you send **closeBlock:**. Any method that accesses information about a block opens it for reading. This means not only does **readBlock:range:** open a block for reading, but so does **sizeOfBlock:**, which returns the block's size. The **copyBlock:** method opens the block for reading, but it also closes it when finished (unless you already had that block opened for reading). Even if you commit a transaction before you send **closeBlock:**, the block remains open for reading.

Any method that changes a block's contents opens the block for writing. This means not only does **openBlock:range:** open a block for writing, but so do the methods **copyBytes:toBlock:range:**, **createBlockOfSize:**, and **freeBlock:**. You indicate that you are finished with a block you have open for writing by having its changes take effect. Closing the block with **closeBlock:** does *not* make your changes take effect, *even if transactions are disabled*. Regardless of whether transactions are enabled or disabled, you must send **commitTransaction** to have your changes actually be made.

If transactions are disabled, **commitTransaction** commits all the changes made to blocks since that last **commitTransaction** or **abortTransaction** message was sent. **abortTransaction** cancels all the changes made since the last **commitTransaction**.

Creating an NSByteStore

+ (NSByteStore *)**byteStore**

Returns a new NSByteStore with transactions disabled.

Managing the NSByteStore

– (unsigned)**count**

Returns the number of blocks in the NSByteStore at transaction level 0. That is, if you have created or freed some blocks but those changes have not been committed at transaction level 0, **count** will not reflect those changes.

– (void)**empty**

Frees all blocks of memory in the NSByteStore. If transactions are enabled, this method starts and commits a new transaction.

– (void)**getBlocks:**(unsigned *)*blocks*

Returns in *blocks* a C-style array of block numbers at transaction level 0. The caller must free the returned array.

– (unsigned)**rootBlock**

Returns the number of the root block, which by convention is used as a table of contents or a directory.

Creating, Copying, and Freeing Blocks

– (unsigned)**createBlockOfSize:**(unsigned)*size*

Returns a block number for a new block of *size* bytes with the contents initialized to zero. Creating a block with size 0 is allowed.

– (unsigned)**copyBlock:**(unsigned)*aBlock*
range:(NSRange)*range*

Returns a block number for a new block whose size and contents are identical to the memory region in block *aBlock* specified by *range*.

– (void)**freeBlock:**(unsigned)*aBlock*

Removes and frees the block *aBlock*.

Opening and Closing Blocks

– (void *)**openBlock:**(unsigned)*aBlock* **range:**(NSRange)*range*
Opens for writing the memory region in block *aBlock* specified by *range*. A pointer to the region is returned.

– (const void *)**readBlock:**(unsigned)*aBlock* **range:**(NSRange)*range*
Opens for reading the memory region in block *aBlock.* specified by *range*. A pointer to the region is returned.

– (void)**closeBlock:**(unsigned)*aBlock*
Closes the block *aBlock*.

Managing Block Sizes

– (void)**resizeBlock:**(unsigned)*aBlock* **toSize:**(unsigned)*size*
Resizes the block *aBlock* to *size* bytes. This method may change the location of the block as well.

– (unsigned)**sizeOfBlock:**(unsigned)*aBlock*
Returns the size in bytes of the block *aBlock*.

Using Transactions

– (unsigned)**startTransaction**
Begins a new transaction, enabling transactions if necessary, for the current context. This transaction will be aborted or committed before all other outstanding transactions. Returns a number that both identifies the new transaction and indicates the number of transactions outstanding.

– (void)**abortTransaction**
Reverts the NSByteStore to the state it was in before the last **startTransaction** message or the last **commitTransaction** message. Any blocks that had been opened are made available to other store contexts.

– (void)**commitTransaction**
Commits all changes made to blocks opened since the last **startTransaction** or the last **commitTransaction** and closes those blocks. If transactions are disabled or the nesting level becomes 0, this method makes all of the changed blocks available to other contexts.

– (BOOL)**areTransactionsEnabled**
Returns YES if transactions are enabled for the NSByteStore, NO if not. Transactions are enabled by the method **startTransaction**.

– (unsigned)**nestingLevel**
Returns the number of transactions pending against the NSByteStore.

– (unsigned)**changeCount**
Returns the number of changes made to the NSByteStore's contents since it was initialized. This number equals the number of **commitTransaction** and **abortTransaction** messages the NSByteStore has received.

Changing the Contents

– (void)**copyBytes:**(const void *)*newData* **toBlock:**(unsigned)*aBlock* **range:**(NSRange)*range*

Copies the series of bytes pointed to by *newData* into the memory region in block *aBlock* specified by *range*. This method will expand the block's size if the data will not fit in the location specified by *range*.

– (NSData *)**contentsAsData**

Creates a virtual memory image of the NSByteStore.

– (void)**replaceContentsWithData:**(NSData *)*data*

Replaces the contents of the NSByteStore with virtual memory image *data*. This method ignores and erases any pending writes to the NSByteStore.

NSByteStoreFile

Inherits From: NSByteStore : NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSByteStore.h

Class Description

NSByteStoreFile is a subclass of NSByteStore that keeps its storage in a file. NSByteStoreFile guarantees the integrity of stored data against process and system crashes when protected by transactions (described in the NSByteStore class specification), provided that the physical media remains intact.

When you create an NSByteStoreFile, you specify a storage file and open it for reading only or for both reading and writing. The methods you use to access the contents of the file are implemented in NSByteStore.

To support the use of preconfigured files, a process using an NSByteStoreFile opened for reading only may freely modify the NSByteStoreFile; all modified pages are reflected only in the address space of the process. The modifications are never written to the file and are discarded when the NSByteStoreFile is freed.

Creating and Initializing an NSByteStoreFile Instance

+ (NSByteStore *)**byteStoreFile:**(NSString*)*path*
transactionsEnabled:(BOOL)*enable*
create:(BOOL)*create*
readOnly:(BOOL)*readOnly*

Creates and initializes an NSByteStoreFile with *path* as its storage file. If *enable* is YES, transactions are enabled. If *create* is YES, the file *path* is created. If *readOnly* is YES, *path* is opened for reading. If *readOnly* is NO, path is opened for reading and writing.

– (id)**initWithPath:**(NSString*)*path*
transactionsEnabled:(BOOL)*enable*
create:(BOOL)*create*
readOnly:(BOOL)*readOnly*

Initializes a newly allocated NSByteStoreFile with *path* as its storage file. If *enable* is YES, transactions are enabled. If *create* is YES, the file *path* is created. If *readOnly* is YES, *path* is opened for reading. If *readOnly* is NO, path is opened for reading and writing.

Accessing the Storage File

– (NSString *)**storePath**

Returns the path of the storage file.

Reducing Memory Consumption

– (void)**compactUntilDate:**(NSDate *)*limitDate*

Removes free space by relocating blocks toward the origin of the virtual address space defined by the NSByteStoreFile. *limitDate* sets a time limit on this operation. No *limitDate* allows the compaction to run to completion.

NSCalendarDate

Inherits From: NSDate : NSObject

Conforms To: NSCoding, NSCopying (NSDate)
NSObject (NSObject)

Declared In: Foundation/NSDate.h

Class Description

NSCalendarDate is a public subclass of NSDate that defines concrete date objects. These objects have time zones and format strings bound to them and are especially suited for representing and manipulating dates according to western calendrical systems.

By drawing on the behavior of the NSTimeZone class, NSCalendarDate objects adjust their visible representations to reflect their associated time zones. Because of this, you can track an NSCalendarDate object across different time zones. You can also present date information from time-zone viewpoints other than the one for the current locale.

Each NSCalendarDate object also has a calendar format string bound to it. This format string contains date-conversion specifiers that are very similar to those used in the standard C library function **strftime()**. By reference to this format string, NSCalendarDate can interpret dates that are represented as strings conforming to the format. Several methods allow you to specify formats other than the one bound to the object, and **setCalendarFormat:** lets you change the default format string for an NSCalendarDate object.

NSCalendarDate provides both class and instance methods for obtaining initialized objects. Some of these methods allow you to initialize date objects from strings while others initialize objects from sets of integers corresponding the standard time values (months, hours, seconds, etc.). As always, you are responsible for deallocating any objects obtained through an **alloc...** or **copy...** method.

To retrieve conventional elements of a date, use the methods of the form **dayOfWeek**, **monthOfYear**, and so on. For example, **dayOfWeek** returns a number that indicates the day of the week (0 is Sunday). The **monthOfYear** method returns a number from 1 to 12 that indicates the month.

NSCalendarDate provides several methods for representing dates as strings. These methods—**description**, **descriptionWithLocale:**, **descriptionWithCalendarFormat**:, and **descriptionWithCalendarFormat:timeZone:**—take an implicit or explicit format string.

NSCalendarDate performs date computations based on western calendar systems, primarily the Gregorian. (The algorithms are derived from public domain software described in “Calendrical Calculations,” a two-part series by Nachum Dershowitz and Edward M. Reingold in *Software—Practice and Experience*).

General Exceptions

NSCalendarDate will raise NSInvalidArgumentException in the general case where numeric character strings to specify years, months, days, and so on, are not valid numbers.

Getting and Initializing an NSCalendar Date

+ (NSCalendarDate *)**calendarDate**

Returns an NSCalendarDate initialized to the current date and time.

+ (NSCalendarDate *)**dateWithString:**(NSString *)*description*
calendarFormat:(NSString *)*format*

Returns an NSCalendarDate object initialized with the date specified in *description* and interpreted according the the conversion specifiers in *format*. Raises NSInvalidArgumentException if the *description* and *format* do not correspond exactly.

+ (NSCalendarDate *)**dateWithString:**(NSString *)*description*
calendarFormat:(NSString *)*format*
locale:(NSDictionary *)*dictionary*

Returns an NSCalendarDate object initialized with the date date specified in *description* and interpreted according the the conversion specifiers in *format*. String components of the date are fetched from the locale *dictionary*. Raises NSInvalidArgumentException if the *description* and *format* do not correspond exactly.

+ (NSCalendarDate *)**dateWithYear:**(int)*year*
month:(unsigned int)*month*
day:(unsigned int)*day*
hour:(unsigned int)*hour*
minute:(unsigned int)*minute*
second:(unsigned int)*second*
timeZone:(NSTimeZone *)*aTimeZone*

Returns an NSCalendarDate object initialized with integers that specify a *year* (which must include the century), *month*, *day*, *hour*, *minute*, and *second*. Also include a time-zone object or time-zone detail object (*aTimeZone*) to have the date adjusted to a particular locale. If you specify **nil** for a time zone, NSInvalidArgumentException is raised. (See "Retrieving Date Elements," below, for the proper ranges of the date and time integers.)

– (id)**initWithString:**(NSString *)*description*

Initializes and returns an NSCalendarDate object specified by *description* in the international format for date representation (YYYY-MM-DD HH:MM:SS ± HHMM, where ± HHMM is an offset from GMT).

– (id)**initWithString:**(NSString *)*description*
calendarFormat:(NSString *)*format*

Initializes and returns an NSCalendarDate object specified as a string object in *description* and interpreted according to the extended **strftime()** date-conversion specifiers in *format*. Raises NSInvalidArgumentException if the *description* and *format* do not correspond exactly.

– (id)**initWithString:**(NSString *)*description* **calendarFormat:**(NSString *)*format* **locale:**(NSDictionary *)*dictionary*

Initializes and returns an NSCalendarDate object specified as a string object in *description* and interpreted according to the extended **strftime** date-conversion specifiers in *format*. String components of the date are fetched from the locale *dictionary*. Raises an NSInvalidArgumentException if the *description* and *format* do not correspond exactly.

– (id)**initWithYear:**(int)*year* **month:**(unsigned int)*month* **day:**(unsigned int)*day* **hour:**(unsigned int)*hour* **minute:**(unsigned int)*minute* **second:**(unsigned int)*second* **timeZone**:(NSTimeZone *)*aTimeZone*

Returns an NSCalendarDate object initialized with integers that specify a *year* (which must include the century), *month*, *day*, *hour*, *minute*, and *second*. Also include a time-zone object (*aTimeZone*) to have the date adjusted for a particular locale. Raises an NSInvalidArgumentException if you specify **nil** for a time zone. (See "RetrievingDate Elements," below, for the proper ranges of the date and time integers.)

Retrieving Date Elements

– (int)**dayOfCommonEra**

Returns the number of days since the beginning of the Common Era.

– (int)**dayOfMonth**

Returns the day of the month (1 through 31) of the NSCalendarDate object.

– (int)**dayOfWeek**

Returns a number indicating the day of the week (0 [Sun] through 6 [Sat]) of the NSCalendarDate object.

– (int)**dayOfYear**

Returns a number indicating the day of the year (1 through 366) of the NSCalendarDate object.

– (int)**hourOfDay**

Returns a number indicating the hour of the day (0 through 23) of the NSCalendarDate object.

– (int)**minuteOfHour**

Returns a number indicating the minute of the hour (0 through 59) of the NSCalendarDate object.

– (int)**monthOfYear**

Returns a number indicating the month of the year (1 through 12) of the NSCalendarDate object.

– (int)**secondOfMinute**

Returns a number indicating the second of the minute (0 through 59) of the NSCalendarDate object.

– (int)**yearOfCommonEra**

Returns a number indicating the year, including the century, of the NSCalendarDate object.

Providing Adjusted Dates

– (NSCalendarDate *)**addYear:**(int)*year*
month:(int)*month*
day:(int)*day*
hour:(int)*hour*
minute:(int)*minute*
second:(int)*second*

Returns an NSCalendarDate objects with the *year*, *month*, *day*, *hour, minute*, and *second* offsets specified as arguments and the correct time-zone detail object for the computed date. These offsets are relative to the object and can be positive or negative. This method preserves "clock time" during transitions to and from Daylight Savings Time and on leap years.

Getting String Descriptions of Dates

– (NSString *)**description**

Returns a string description of the receiver's date using the default format string (%Y-%m-%d %H:%M:%S %z) and the locale and time-zone information associated with the receiver.

– (NSString *)**descriptionWithCalendarFormat:**(NSString *)*format*

Returns a string description of the receiver's date that is formatted according to the conversion specifiers in *format* and using the locale and time-zone detail information associated with the receiver.

– (NSString *)**descriptionWithCalendarFormat:**(NSString *)*format*
locale:(NSDictionary *)*locale*

Returns a string description of the receiver's date that is formatted according to the conversion specifiers in *format*, represented according to the locale information in *locale*, and adjusted according to the time-zone detail information associated with the receiver.

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns a string description of the receiver's date using the default format string (%Y-%m-%d %H:%M:%S %z), with information localized according to the locale information in *locale*, and using the time zone information associated with the receiver.

Getting and Setting Calendar Formats

– (NSString *)**calendarFormat**

Returns the calendar format (a string of date-conversion specifiers) for the receiving object. The default calendar format is "%Y-%m-%d %H:%M:%S %z".

– (void)**setCalendarFormat:**(NSString *)*format*

Sets the calendar format for the receiving object to *format*.

Getting and Setting Time Zones

– (void)**setTimeZone:**(NSTimeZone *)*aTimeZone* Sets the time-zone object associated with the NSCalendarDate object to *aTimeZone*.

– (NSTimeZoneDetail *)**timeZoneDetail** Returns the NSTimeZoneDetail object associated with the receiver.

NSCharacterSet

Inherits From: NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying
NSObject (NSObject)

Declared In: Foundation/NSCharacterSet.h

Class Description

The NSCharacterSet class declares the programmatic interface to objects that construct immutable *descriptions* of character sets in the Unicode character encoding. Using NSCharacterSet objects, you can determine if a given Unicode character belongs to a specified set. See NSMutableCharacterSet for a class that constructs descriptions of character sets that can be modified dynamically. NSCharacterSet's primitive methods are **characterIsMember:** and **bitmapRepresentation**. Subclasses of NSCharacterSet must implement these two methods.

NSCharacterSet objects can be thought of as loosely analogous to the **is…** macros (such as **isupper()**)available in the **ctype** collection of most standard C libraries. NSCharacterSet objects, however, offer much greater flexibility in that you can dynamically construct your own custom character sets against which you can test characters.

The term "bitmap" in the descriptions below does not refer to "bitmap characters" in the sense of screen fonts for display. The "bitmaps" referred to here are compact ordered *bit set* representations of Unicode character positions or ranges of Unicode characters.

You create "standard" character sets—such as a set of alphanumerics, or a set of decimal digits—by invoking the NSCharacterSet class object with one of the methods described in "Creating a Standard Character Set". These methods provide convenient means to create a standard set without needing to specify the character positons explicitly.

You can also create your own "custom" character sets by using one of the methods described under "Creating a Custom Character Set". To create a character set with multiple disjoint ranges, see the **add**… methods described in NSMutableCharacterSet.

Creating a Standard Character Set

+ (NSCharacterSet *)**alphanumericCharacterSet** Returns a character set containing the uppercase and lowercase alphabetic characters (a–z, A–Z, other alphabetic characters such as é, É, ç, Ç, and so on) and the decimal digit characters (0–9).

+ (NSCharacterSet *)**controlCharacterSet** Returns a character set containing the control characters (characters with decimal Unicode values 0 to 31 and 127 to 159).

+ (NSCharacterSet *)**decimalDigitCharacterSet**
Returns a character set containing only decimal digit characters (0–9).

+ (NSCharacterSet *)**decomposableCharacterSet**
Returns a character set containing all individual Unicode characters that can also be represented as composed character sequences.

+ (NSCharacterSet *)**illegalCharacterSet**
Returns a character set containing the illegal Unicode values.

+ (NSCharacterSet *)**letterCharacterSet**
Returns a character set containing the uppercase and lowercase alphabetic characters (a–z, A–Z, other alphabetic characters such as é, É, ç, Ç, and so on).

+ (NSCharacterSet *)**lowercaseLetterCharacterSet**
Returns a character set containing only lowercase alphabetic characters (a–z, other alphabetic characters such as é, ç, and so on).

+ (NSCharacterSet *)**nonBaseCharacterSet**
Returns a set containing all characters which are not defined to be base characters for purposes of dynamic character composition.

+ (NSCharacterSet *)**uppercaseLetterCharacterSet**
Returns a character set containing only uppercase alphabetic characters (A–Z, other alphabetic characters such as É, Ç, and so on).

+ (NSCharacterSet *)**whitespaceAndNewlineCharacterSet**
Returns a character set containing only whitespace characters (space and tab) and the newline character.

+ (NSCharacterSet *)**whitespaceCharacterSet**
Returns a character set containing only in-line whitespace characters (space and tab). This set doesn't contain the newline or carriage return characters.

Creating a Custom Character Set

+ (NSCharacterSet *)**characterSetWithBitmapRepresentation:**(NSData *)*data*
Returns a character set containing characters determined by the bitmap representation *data*.

+ (NSCharacterSet *)**characterSetWithCharactersInString:**(NSString *)*aString*
Returns a character set containing the characters in *aString*. If *aString* is empty, an empty character set is returned. *aString* must not be **nil**.

+ (NSCharacterSet *)**characterSetWithRange:**(NSRange)*aRange*

Returns a character set containing characters whose Unicode values are given by *aRange*.

Getting a Binary Representation

– (NSData *)**bitmapRepresentation**

Returns an NSData object encoding the receiving character set in binary format. This format is suitable for saving to a file or otherwise transmitting or archiving.

Testing Set Membership

– (BOOL)**characterIsMember:**(unichar)*aCharacter*

Returns YES if *aCharacter* is in the receiving character set, NO if it isn't.

Inverting a Character Set

– (NSCharacterSet *)**invertedSet**

Returns a character set containing only characters that *don't* exist in the receiver.

NSCoder

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSCoder.h
Foundation/NSGeometry.h

Class Description

NSCoder is an abstract class that declares the interface used by subclasses to take objects from dynamic memory and code them into and out of some other format. This capability provides the basis for archiving (where objects and other structures are stored on disk) and distribution (where objects are copied to different address spaces). See the NSArchiver and NSUnarchiver class specifications for more information on archiving.

NSCoder operates on the basic C and Objective C types—**int**, **float**, **id**, and so on (but excluding **void *** and **union**)—as well as on user-defined structures and pointers to these types.

NSCoder declares methods that a subclass can override if it wants:

- To encode or decode an object only under certain conditions, such as it being an intrinsic part of a larger structure (**encodeRootObject:** and **encodeConditionalObject:**).
- To allow decoded objects to be allocated from a specific memory zone (**setObjectZone:**).
- To allow system versioning (**systemVersion**).

NSCoder differs from the NSSerializer and NSDeserializer classes in that NSCoders aren't restricted to operating on property list objects (objects of the NSData, NSString, NSArray, and NSDictionary classes). Also, unlike NSSerializers, NSCoders store type information along with the data. Thus, an object decoded from a stream of bytes will be of the same class as the object that was originally encoded into the stream.

Encoding and Decoding Objects

In OpenStep, coding is facilitated by methods declared in several places, most notably the NSCoder class, the NSObject class, and the NSCoding protocol.

The NSCoding protocol declares the two methods (**encodeWithCoder:** and **initWithCoder:**) that a class must implement so that objects of that class can be encoded and decoded. When an object receives an **encodeWithCoder:** message, it should send a message to **super** to encode inherited instance variables before it encodes the instance variables that it's class declares. For example, a fictitious MapView class that displays a legend and a map at various magnifications, might implement **encodeWithCoder:** like this:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    [coder encodeValuesOfObjCTypes:"si@", &mapName, &magnification, &legendView];
}
```

Objects are decoded in two steps. First, an object of the appropriate class is allocated and then it's sent an **initWithCoder:** messages to allow it to initialize its instance variables. Again, the object should first send a message to **super** to initialized inherited instance variables, and then it should initialize its own. MapView's implementation of this method looks like this:

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    [coder decodeValuesOfObjCTypes:"si@", &mapName, &magnification, &legendView];
    return self;
}
```

Note the assignment of the return value of **initWithCoder:** to **self** in the example above. This is done in the subclass because the superclass, in its implementation of **initWithCoder:**, may decide to return a object other than itself.

There are other methods that allow an object to customize its response to encoding or decoding. NSObject declares these methods:

Method	Typical Use
classForCoder:	Allows an object, when being encoded, to substitute a class other than its own. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.
replacementObjectForCoder:	Allows an object, when being encoded, to substitute another object for itself. For example, an object might encode itself into an archive, but encode a proxy for itself if it's being encoded for distribution.
awakeAfterUsingCoder:	Allows an object, when being decoded, to substitute another object for itself. For example, an object that represents a font might, upon being decoded, release itself and return an existing object having the same font description as itself. In this way, redundant objects can be eliminated.

See the NSObject class specification for more information.

Encoding Data

– (void)**encodeArrayOfObjCType:**(const char *)*types*
count:(unsigned int)*count*
at:(const void *)*array*

Encodes data of Objective C types listed in *types* having *count* elements residing at address *array.*

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

Overridden by subclasses to encode the supplied Objective C object so that a copy rather than a proxy of *anObject* is created upon decoding. NSCoder's implementation simply invokes **encodeObject:**.

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

Overridden by subclasses to conditionally encode the supplied Objective C object. The object should be encoded only if it is an intrinsic member of the larger data structure. NSCoder's implementation simply invokes **encodeObject:**.

– (void)**encodeDataObject:**(NSData *)*data*

Encodes the NSData object *data*.

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

Encodes the supplied Objective C object.

– (void)**encodePropertyList:**(id)*aPropertyList*

Encodes the supplied property list (NSData, NSArray, NSDictionary, or NSString objects).

– (void)**encodePoint:**(NSPoint)*point*

Encodes the supplied point structure.

– (void)**encodeRect:**(NSRect)*rect*

Encodes the supplied rectangle structure.

– (void)**encodeRootObject:**(id)*rootObject*

Overridden by subclasses to start encoding an interconnected group of Objective C objects, starting with *rootObject*. NSCoder's implementation simply invokes **encodeObject:**.

– (void)**encodeSize:**(NSSize)*size*

Encodes the supplied size structure.

– (void)**encodeValueOfObjCType:**(const char *)*type*
at:(const void *)*add*ress

Encodes data of the specified Objective C type residing at *address*.

– (void)**encodeValuesOfObjCTypes:**(const char *)*types*,...

Encodes values corresponding to the Objective C types listed in *types* argument list.

Decoding Data

– (void)**decodeArrayOfObjCType:**(const char *)*types*
count:(unsigned)*count*
at:(void *)*address*

Decodes data of Objective C types listed in *type* having *count* elements residing at *address*.

– (NSData *)**decodeDataObject**

Decodes and returns an NSData object.

– (id)**decodeObject**

Decodes an Objective C object.

– (id)**decodePropertyList** Decodes a property list (NSData, NSArray, NSDictionary, or NSString objects).

– (NSPoint)**decodePoint** Decodes a point structure.

– (NSRect)**decodeRect** Decodes a rectangle structure.

– (NSSize)**decodeSize** Decodes a size structure.

– (void)**decodeValueOfObjCType:**(const char *)*type* **at:**(void *)*address* Decodes data of the specified Objective C *type* residing at *address*. You are responsible for releasing the resulting objects.

– (void)**decodeValuesOfObjCTypes:**(const char *)*types*,... Decodes values corresponding to the Objective C types listed in *types* argument list. You are responsible for releasing the resulting objects.

Managing Zones

– (NSZone *)**objectZone** Returns the memory zone used by decoded objects. For instances of NSCoder, this is the default memory zone, the one returned by **NSDefaultMallocZone()**.

– (void)**setObjectZone:**(NSZone *)*zone* Sets the memory zone used by decoded objects. Instances of NSCoder always use the default memory zone, the one returned by **NSDefaultMallocZone()**, and so ignore this method.

Getting a Version

– (unsigned int)**systemVersion** Returns the system version number as of the time the archive was created.

– (unsigned int)**versionForClassName:**(NSString *)*className* Returns the version number of the class *className* as of the time it was archived.

NSConditionLock

Inherits From: NSObject

Conforms To: NSLocking
NSObject (NSObject)

Declared In: Foundation/NSLock.h

Class Description

NSConditionLock objects are used to lock and unlock threads when specified conditions occur.

The user of an NSConditionLock object can lock when a process enters a particular state and can set the state to something else when releasing the lock. The states are defined by the lock's user. NSConditionLock is well suited to synchronizing different modules such as a producer and a consumer where the two modules must share data, but the consumer must sleep until a condition is met such as more data being available.

The NSConditionLock class provides four ways of locking its objects (**lock**, **lockWhenCondition:**, **tryLock**, and **tryLockWhenCondition**) and two ways of unlocking (**unlock** and **unlockWithCondition:**). Any combination of locking method and unlocking method is legal.

The following example shows how the producer-consumer problem might be handled using condition locks. The producer need not wait for a condition, but must wait for the lock to be made available so it can safely create shared data. For example, a producer could use a lock this way:

```
/* create the lock only once */
id condLock = [NSConditionLock new];

[condLock lock];
/* Manipulate global data... */
[condLock unlockWithCondition:HAS_DATA];
```

Multiple consumer threads can then lock until there's data available and everyone is out of locked critical sections. In the following code sample, the consumer sleeps until the producer invokes **unlockWithCondition:** with the parameter HAS_DATA:

```
[condLock lockWhenCondition:HAS_DATA];
/* Manipulate global data if necessary... */
[condLock unlockWithCondition:(moreData ? HAS_DATA : NO_DATA)];
```

An NSConditionLock object doesn't busy-wait, so it can be used to lock time-consuming operations without degrading system performance.

The NSConditionLock, NSLock, and NSRecursiveLock classes all implement the NSLocking protocol with various features and performance characteristics; see the other class descriptions for more information.

Initializing an NSConditionLock

– (id)**initWithCondition:**(int)*condition*	Initializes a newly created NSConditionLock and sets its condition to *condition*.

Returning the Condition

– (int)**condition**	Returns the receiver's condition, the state that must be achieved before a conditional lock can be acquired or released.

Acquiring and Releasing a Lock

– (void)**lockWhenCondition:**(int)*condition*	Attempts to acquire a lock when *condition* is met. Blocks until *condition* is met.
– (void)**unlockWithCondition:**(int)*condition*	Releases the lock and sets lock state to *condition*.
– (BOOL)**tryLock**	Attempts to acquire a lock. Returns YES if successful and NO otherwise.
– (BOOL)**tryLockWhenCondition:**(int)*condition*	Attempts to acquire a lock when *condition* is met. Returns YES if successful and NO otherwise.

NSConnection

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSConnection.h

Class Description

The NSConnection class declares the programmatic interface to objects that manage a *connection* such that objects in one thread can send messages to objects in another thread (typically, in another process, and it defines instances that manage each side of such a connection.

Each distinct thread of execution has one default connection defined. Any given thread can have as many connections as desired, but a given connection can be served by only one thread.

To set up a connection, some object in your application must be established as what is known as a "root" object and registered with a name in the Network Name Server. Such root objects can then be connected to by other threads, and can receive messages sent to them from other threads. An easy way to establish an object as a root object is to send the **defaultConnection** method to the NSConnection class object to obtain a connection object. Then use **setRootObject:** to establish the desired object as the object that will be registered, followed by **registerName:** to make that object available to the Network Name Server under the specified name.

To obtain a connection to an object registered elsewhere, you will generally send the **rootProxyForConnectionWithRegisteredName:host:** method to the NSConnection class object. This method returns a proxy to the remote object. You should then inform the proxy of the protocol(s) the remote object responds to using **setProtocolForProxy:**. To obtain the actual connection object instead of the proxy, use the **connectionWithRegisteredName:host:** method.

If the string **@"*"** is used where a hostname is required, it implies a lookup for any server registered with the specified name on the local subnet. If **nil** is supplied where a hostname is required, the name lookup occurs only on the local host.

When an NSConnection object is deallocated, the notification NSConnectionDeath is posted to the default notification center with that NSConnection object.

Exceptions

NSConnection can raise NSInternalInconsistencyException for a variety of reasons when it detects "impossible" situations. In addition, NSConnection can raise NSInvalidArgumentException when a remote method invocation sends an unknown selector.

Initializing a Connection

– (id)**init** — Initialize a newly allocated NSConnection suitable for a new registry and new name.

Establishing a Connection

+ (NSConnection *)**connectionWithRegisteredName:**(NSString *)*name* **host:**(NSString *)*hostName* — Registers and returns a connection with *name* on *hostName*.

+ (NSConnection *)**defaultConnection** — Establishes and returns a default per-thread connection.

+ (NSDistantObject *)**rootProxyForConnectionWithRegisteredName:**(NSString *)*name* **host**:(NSString *)*hostName* — Registers a connection with *name* on *hostName* and returns its root proxy.

Determining Connections

+ (NSArray *)**allConnections** — Returns an array describing all existing valid connections.

– (BOOL)**isValid** — Identifies that the receiver is a valid connection.

Registering a Connection

– (BOOL)**registerName:**(NSString *)*name* — Registers the connection with *name* on the local system and returns YES if the registration was successful, NO otherwise.

Assigning a Delegate

– (id)**delegate** — Returns the connection's delegate.

– (void)**setDelegate:**(id)*anObject* — Sets the connection's delegate.

Getting and Setting the Root Object

– (id)**rootObject** — Returns the root object served.

– (NSDistantObject *)**rootProxy** — Returns an NSDistantObject proxy to the root object served by this connection.

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

Sets the root object being served to *anObject*; if the root object already exists, replaces it with *anObject*. Be aware that if the root object is replaced while a connection is active, existing root proxies on the client side of the connection will continue to communicate with the previous root object, while new proxies will communicate with the newly established root object.

Request Mode

– (NSString *)**requestMode**

Returns the mode in which requests are honored.

– (void)**setRequestMode:**(NSString *)*mode*

Sets the mode in which requests are honored to *mode*.

Conversation Queueing

–(BOOL)**independentConversationQueueing** Returns **conversationQueuing** mode. The default value is NO.

–(void)**setIndependentConversationQueueing:**(BOOL)*flag*
If *flag* is YES, unrelated requests are queued for later processing. This allows a server to use distributed objects freely in its implementation without concern for the consistency of its internal state. Note that this can cause deadlocks among peers.

Timeouts

– (NSTimeInterval)**replyTimeout** Returns the reply timeout time interval.

– (NSTimeInterval)**requestTimeout** Returns the request timeout time interval.

– (void)**setReplyTimeout:**(NSTimeInterval)*interval* Sets the reply timeout to the time interval *interval*.

– (void)**setRequestTimeout:**(NSTimeInterval)*interval* Sets the request timeout to the time interval *interval*.

Get Statistics

– (NSDictionary *)**statistics** Returns statistics for this connection.

Implemented by the Delegate

– (BOOL)**makeNewConnection:**(NSConnection *)*connection*
sender:(NSConnection *)*ancestor* Asks permission to create a new connection *connection* where *ancestor* is the ancestral connection; returns YES if connection allowed.

NSCountedSet

Inherits From: NSMutableSet : NSSet : NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying (NSSet)
NSObject (NSObject)

Declared In: Foundation/NSSet.h

Class Description

The NSCountedSet class declares the programmatic interface to an object that manages a mutable set of objects. NSCountedSet provides support for the mathematical concept of a *counted set*. A counted set, both in its mathematical sense and in the OpenStep implementation of NSCountedSet, is an unordered collection of elements, just as in a regular set, but the elements of the set aren't necessarily distinct. In the literature, a counted set is also knownas a *bag*.

Each new—that is, distinct—object inserted into an NSCountedSet object has a counter associated with it. NSCountedSet keeps track of the number of times objects are inserted and requires that objects are removed the same number of times. OpenStep also provides the NSSet class for sets whose elements *are* distinct—that is, there is only one instance of an object in an NSSet even if the object has been added to the set multiple times.

Use set objects as an alternative to array objects when the order of elements is not important, but performance in testing whether an object is contained in the set *is* a consideration—while arrays are ordered, testing for membership is slower than with sets.

Objects in a set must respond to **hash** and **isEqual:** methods. See the NSObject protocol for details on **hash** and **isEqual:**. Each new distinct object must provide a unique hash value.

Generally, you instantiate an NSCountedSet object by sending one of the **set…** methods to the NSCountedSet class object, as described in NSSet. These methods return an NSCountedSet object containing the elements (if any) you pass in as arguments. Newly created instances of NSCountedSet created by invoking the **set** method can be populated with objects using any of the **init…** methods. **initWithObjects::** is the designated initializer for this class.

You add or remove objects from a counted set using the **addObject:** and **removeObject:** methods.

An NSCountedSet may be queried using the **objectEnumerator** method, which provides for traversing elements of the set one by one. The **countForObject:** method returns the number of times the specified object has been added to this set.

Initializing an NSCountedSet

– (id)**initWithArray:**(NSArray *)*anArray*

Initializes a newly allocated set object by placing in it the objects contained in *anArray.*

– (id)**initWithCapacity:**(unsigned int)*numItems*

Initializes a newly allocated set object, giving it enough memory to hold *numItems* objects.

– (id)**initWithSet:**(NSSet *)*anotherSet*

Initializes a newly allocated set object by placing in it the objects contained in *anotherSet.*

Adding Objects

– (void)**addObject:**(id)*anObject*

Adds *anObject* to the set, unless *anObject* is equal to some object already in the set. In either case, the counter that's returned by **countForObject:** is incremented.

Removing Objects

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

Decrements the counter for the object, if the set contains an object that's equal to *anObject*. If this causes the counter to reach zero, the object that's equal to *anObject* is removed from the set.

Querying the NSCountedSet

– (unsigned int)**countForObject:**(id)*anObject*

Returns the number of times that an object equal to *anObject* has ostensibly been added to the set. (This number is incremented by **addObject:** and decremented by **removeObject:**.)

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that will access each object in the set only once, regardless of its count.

NSData

Inherits From: NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying
NSObject (NSObject)

Declared In: Foundation/NSData.h

Class Description

The NSData class declares the programmatic interface to objects that contain data in the form of bytes. NSData objects hold a static collection of bytes; NSData's subclass, NSMutableData, defines objects that hold modifiable data. These two classes provide an object-oriented approach to memory allocation, a facility that in procedural programming is accessed through functions like **malloc()**. Furthermore, these classes take advantage of operating system primitives when allocating large blocks of memory.

NSData's two primitive methods—**bytes** and **length**—provide the basis for all the other methods in its interface. The **bytes** method returns a pointer to the bytes contained in the data object. **length** returns the number of bytes contained in the data object.

NSData and NSMutableData objects are commonly used to hold the contents of a file. The methods **dataWithContentsOfFile:** and **dataWithContentsOfMappedFile:** return objects that represent a file's contents. The **writeToFile:atomically:** method enables you to write the contents of a data object to a file.

NSData provides access methods for copying bytes from a data object into a buffer. Use **getBytes:** to copy the entire contents of the object or **getBytes:length:** to copy a subset, starting with the first byte. **getBytes:range:** copies a range of bytes from a starting point within the bytes themselves. You can also return a data object that contains a subset of the bytes in another data object by using the **subdataWithRange:** method. Or, you can use the **description** method to return an NSString representation of the bytes in a data object.

For determining if two data objects are equal, NSData provides the **isEqualToData:** method, which does a byte-for-byte comparison.

Allocating and Initializing an NSData Object

+ (id)**allocWithZone:**(NSZone *)*zone*	Creates and returns an uninitialized object from *zone*.
+ (id)**data**	Creates and returns an empty object. This method is declared primarily for mutable subclasses of NSData.
+ (id)**dataWithBytes:**(const void *)*bytes* **length:**(unsigned int)*length*	Creates and returns an object containing *length* bytes of data copied from the buffer *bytes*.
+ (id)**dataWithBytesNoCopy:**(void *)*bytes* **length:**(unsigned int)*length*	Creates and returns an object containing *length* bytes from the buffer *bytes*.

+ (id)**dataWithContentsOfFile:**(NSString *)*path* — Creates and returns an object by reading data from the file specified by *path*.

+ (id)**dataWithContentsOfMappedFile:**(NSString *)*path* — Creates and returns an object whose contents come from the mapped file *path*, assuming mapped files are available on the underlying operating system. If mapped files are not available, this method is identical to **dataWithContentsOfFile:**.

– (id)**initWithBytes:**(const void *)*bytes* **length:**(unsigned int)*length* — Initializes a newly allocated NSData object by putting in it *length* bytes of data copied from the buffer bytes.

– (id)**initWithBytesNoCopy:**(void *)*bytes* **length:**(unsigned int)*length* — Initializes a newly allocated NSData object by putting in it *length* bytes of data from the buffer bytes.

– (id)**initWithContentsOfFile:**(NSString *)*path* — Initializes a newly allocated NSData object by reading into it the data from the file specified by *path*.

– (id)**initWithContentsOfMappedFile:**(NSString *)*path* — Initializes a newly allocated NSData object to contain the data residing in the mapped file *path*, assuming mapped files are available on the underlying operating system. If mapped files are not available, this method is identical to **initWithContentsOfFile:**.

– (id)**initWithData:**(NSData *)*data* — Initializes a newly allocated NSData object by placing in it the contents of another NSData object, *data*.

Accessing Data

– (const void *)**bytes** — Returns a pointer to the object's contents. This method returns read-only access to the data.

– (NSString *)**description** — Returns an NSString object that contains a hexadecimal representation of the the receiver's contents.

– (void)**getBytes:**(void *)*buffer* — Copies the receiver's contents into *buffer*.

– (void)**getBytes:**(void *)*buffer* **length:**(unsigned int)*length* — Copies length bytes of the receiver's contents into *buffer*.

– (void)**getBytes:**(void *)*buffer* **range:**(NSRange)aRange — Copies into *buffer* the portion of the receiver's contents within *aRange*. Raises an NSRangeException if *aRange* is not within the range of the receiver's data.

– (NSData *)**subdataWithRange:**(NSRange)*aRange* — Returns an object containing a copy of the receiver's bytes that fall within the limits specified by *aRange*. Raises an NSRangeException if *aRange* is not within the range of the receiver's data.

Querying a Data Object

– (BOOL)**isEqualToData:**(NSData *)*other*

Compares the receiving object to other. If the contents of *other* are equal to the contents of the receiver, this method returns YES. If not, it returns NO.

– (unsigned int)**length**

Returns the number of bytes contained in the receiver.

Storing Data

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*useAuxiliaryFile*

Writes the bytes in the receiving object to the file specified by *path*. If *useAuxiliaryFile* is YES, the data is written to a backup file and then, assuming no errors occur, the backup file is renamed atomically to the intended file name.

Deserializing Data

– (unsigned int)**deserializeAlignedBytesLengthAtCursor:**(unsigned int*)*cursor*

Returns the length of the serialized bytes at the location referenced by *cursor*. If the bytes have been page-aligned, it also obtains the relevant “hole” information and adjusts the cursor. An invocation of this method must have a corresponding **serializeAlignedBytesLength:** invocation.

– (void)**deserializeBytes:**(void *)*buffer* **length:**(unsigned int)*bytes* **atCursor:**(unsigned int*)*cursor*

Deserializes *bytes* number of bytes in the buffer pointed at by *buffer*, places them internally starting at *cursor*, and advances the cursor.

– (void)**deserializeDataAt:**(void *)*data* **ofObjCType:**(const char *)*type* **atCursor:**(unsigned int*)*cursor* **context:**(id <NSObjCTypeSerializationCallBack>) *callback*

Deserializes the data pointed at by *cursor*, interpreting it by the Objective C type specifier *type* and writing it to the memory location referenced by *data*. If the data element is an object other than an instance of NSDictionary, NSArray, NSString, or NSData, a callback from object *callback* can provide further definition of the object. All Objective C types are currently supported except **union** and **void ***. Pointers refer to a single item.

– (int)**deserializeIntAtCursor:**(unsigned int*)*cursor* Deserializes and returns the integer encoded at *cursor.* Also advances the cursor.

– (int)**deserializeIntAtIndex:**(unsigned int)*index* Deserializes and returns the integer encoded at offset *index*. Does not advance the cursor.

– (void)**deserializeInts:**(int *)*intBuffer*
count:(unsigned int)*numInts*
atCursor:(unsigned int*)*cursor*
Deserializes *numInts* integers encoded at the location referenced by *cursor* and puts them in the buffer *intBuffer.* Also advances the cursor.

– (void)**deserializeInts:**(int *)*intBuffer*
count:(unsigned int)*numInts*
atIndex:(unsigned int)*index*
Deserializes *numInts* integers encoded at offset *index* and puts them in the buffer *intBuffer.* Does not advance the cursor.

NSDate

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: Foundation/NSDate.h

Class Description

NSDate is an abstract class that provides behavior for creating dates, comparing dates, representing dates, computing time intervals, and similar functionality. It presents a programmatic interface through which suitable date objects are requested and returned. NSDate objects are lightweight and immutable since they represent a invariant point in time. This class is designed to provide the foundation for arbitrary calendrical representations. Its subclass NSCalendarDate offers date objects that are suitable for representing dates according to western calendrical systems.

“Date” as used above implies clock time as well. The standard unit of time for date objects is a value typed as NSTimeInterval (a **double**) and expressed as seconds. The NSTimeInterval type makes possible a wide and fine-grained range of date and time values, giving accuracy within milliseconds for dates 10,000 years apart.

NSDate and its subclasses compute time as seconds *relative* to an absolute reference date. This reference date is the first instant of January 1, 2001. NSDate converts all date and time representations to and from NSTimeInterval values that are relative to this absolute reference date. A positive interval relative to a date represents a point in the future, a negative interval represents a time in the past.

Note: Conventional UNIX systems implement time according to the Network Time Protocol (NTP) standard, which is based on Coordinated Universal Time. The private implementation of NSDate follows the NTP standard. However, this standard doesn’t account for leap seconds and therefore isn’t synchronized with International Atomic Time (the most accurate).

Like various other Foundation classes, NSDate lets you obtain operating-system functionality (dates and times) without depending on operating-system internals. It also provides a basis for the NSRunLoop and NSTimer classes, which use concrete date objects to implement local event loops and timers.

NSDate’s sole primitive method, **timeIntervalSinceReferenceDate**, provides the basis for all the other methods in the NSDate interface. It returns a time value relative to an absolute reference date.

Using NSDate

The date objects dispensed by NSDate give you a diverse range of date and time functionality. To obtain dates, send one of the **date...** messages to the NSDate class object. One of the most useful is **date** itself, which returns a date object representing the current date and time. You can get new date objects with date and time values adjusted from existing date objects by sending **addTimeInterval:**.

You can obtain relative date information by sending the **timeInterval...** messges to a date object. For instance, **timeIntervalSinceNow** gives you the time, in seconds, between the current time and the receiving date object. Compare dates with the **isEqual:**, **compare:**, **laterDate:**, and **earlierDate:** methods and use the **description** method to obtain a string object that represents the date in a standard international format.

Creating an NSDate Object

+ (id)**allocWithZone:**(NSZone *)*zone*
: Allocates an unitialized NSDate in *zone*. Returns **nil** if allocation fails.

+ (NSDate *)**date**
: Creates and returns an NSDate set to the current date and time.

+ (NSDate *)**dateWithTimeIntervalSinceNow:**(NSTimeInterval)*seconds*
: Creates and returns an NSDate set to *seconds* seconds from the current date and time.

+ (NSDate *)**dateWithTimeIntervalSince1970:**(NSTimeInterval)*seconds*
: Creates and returns an NSDate set to to *seconds* seconds from the reference date used by UNIX® systems. Use a negative argument value to specify a date and time before the reference date.

+ (NSDate *)**dateWithTimeIntervalSinceReferenceDate:**(NSTimeInterval)*seconds*
: Creates and returns an NSDate set to *seconds* seconds from the absolute reference date (the first instant of 1 January, 2001). Use a negative argument value to specify a date and time before the reference date.

+ (NSDate *)**distantFuture**
: Creates and returns an NSDate that represents a date in the distant future (in terms of centuries). You can use this object in your code as a control date, a guaranteed outer temporal limit.

+ (NSDate *)**distantPast**
: Creates and returns an NSDate that represents a date in the distant past (in terms of centuries). You can use this object in your code as a control date, a guaranteed temporal boundary.

– (id)**init**
: Initializes a newly allocated NSDate to the current date and time.

– (id)**initWithString:**(NSString *)*description*
: Returns an NSDate with a date and time value specified by the international string-representation format: YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM is a time zone offset in hours and minutes from Greenwich Mean Time.

– (NSDate *)**initWithTimeInterval:**(NSTimeInterval)*seconds*
sinceDate:(NSDate *)*anotherDate*

Returns an NSDate initialized relative to another date object by *seconds* (plus or minus).

– (NSDate *)**initWithTimeIntervalSinceNow:**(NSTimeInterval)*seconds*

Returns an NSDate initialized relative to the current date and time by *seconds* (plus or minus).

– (id)**initWithTimeIntervalSinceReferenceDate:**(NSTimeInterval)*seconds*

Returns an NSDate initialized relative to the reference date and time by *seconds* (plus or minus).

Converting to an NSCalendar Object

– (NSCalendarDate *)**dateWithCalendarFormat:**(NSString *)*formatString*
timeZone:(NSTimeZone *)*timeZone*

Returns an NSCalendarDate object bound to the format string *formatString* and the time zone *timeZone*. If you specify **nil** after either or both of these arguments, the default format string and time zone are assumed.

Representing Dates

– (NSString *)**description**

Returns a string representation of the receiver. The representation conforms to the international format YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM represents the time-zone offset in hours and minutes from Greenwich Mean Time (GMT).

– (NSString *)**descriptionWithCalendarFormat:**(NSString *)*formatString*
timeZone:(NSTimeZone *)*aTimeZone*
locale:(NSDictionary *)*localeDictionary*

Returns a string representation of the receiver. The representation conforms to *formatString* (a **strftime**-style date-conversion string) and is adjusted to *aTimeZone*. Included are the keys and values that represent the locale data from *localeDictionary.*

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary*

Returns a string representation of receiver (see **description**). Included are the key and values that represent the locale data from *localeDictionary.*

Adding and Getting Intervals

+ (NSTimeInterval)**timeIntervalSinceReferenceDate**

Returns the interval between the system's absolute reference date and the current date and time. This value is less than zero until the first instant of 1 January 2001.

– **addTimeInterval:**(NSTimeInterval)*seconds*

Returns an NSDate that's set to a specified number of seconds relative to the receiver.

– (NSTimeInterval)**timeIntervalSince1970**

Returns the interval between the receiver and the reference date used by UNIX® systems.

– (NSTimeInterval)**timeIntervalSinceDate:**(NSDate *)*anotherDate*

Returns the interval between the receiver and *anotherDate*.

– (NSTimeInterval)**timeIntervalSinceNow**

Returns the interval between the receiver and the current date and time.

– (NSTimeInterval)**timeIntervalSinceReferenceDate**

Returns the interval between the receiver and the system's absolute reference date. This value is less than zero until the first instant of 1 January 2001.

Comparing Dates

– (NSComparisonResult)**compare:**(NSDate *)*anotherDate*

Compares the receiver's date to that of *anotherDate* and returns NSOrderedDescending if the receiver is temporally later, NSOrderedAscending if it's temporally earlier, and NSOrderedSame if they are equal.

– (NSDate *)**earlierDate:**(NSDate *)*anotherDate*

Compares the receiver's date to *anotherDate* and returns the one that's temporally earlier.

– (BOOL)**isEqual:**(id)*anotherDate*

Returns YES if *anotherDate* and the receiver are within one second of each other; otherwise, returns NO.

– (NSDate *)**laterDate:**(NSDate *)*anotherDate*

Compares the receiver's date to *anotherDate* and returns the one that's temporally later.

NSDeserializer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSSerialization.h

Class Description

The NSDeserializer class declares methods that convert an abstract representation of a property list (as contained in an NSData object) into a graph of property list objects in memory. The NSDeserializer class object itself provides these methods; you don't create instances of NSDeserializer. Options to these methods allow you to specify that container objects (arrays or dictionaries) in the resulting graph be mutable or immutable; that deserialization begin at the start of the data or from some position within it; or that deserialization occur lazily, so that a property list is deserialized only if it is actually going to be accessed. See the NSSerializer specification for more information on serialization.

Deserialization Into Property Lists

+ (id)**deserializePropertyListFromData:**(NSData *)*data*
atCursor:(unsigned int*)*cursor*
mutableContainers:(BOOL)*mutable*

Returns a property list object corresponding to the abstract representation in *data* at the location *cursor*. If *mutable* is YES and the object is a dictionary or an array, the re-composed object is made mutable. Returns **nil** if the object is not a valid one for property lists.

+ (id)**deserializePropertyListFromData:**(NSData *)*data*
mutableContainers:(BOOL)*mutable*

Returns a property list object corresponding to the abstract representation in *data* or **nil** if *data* doesn't represent a property list. If *mutable* is YES and the object is a dictionary or an array, the re-composed object is made mutable.

+ (id)**deserializePropertyListLazilyFromData:**(NSData *)*data*
atCursor:(unsigned int*)*cursor*
length:(unsigned int)*length*
mutableContainers:(BOOL)*mutable*

Returns a property list from *data* at location *cursor* or **nil** if *data* doesn't represent a property list. The deserialization proceeds lazily. That is, if *data* at *cursor* has a length greater than *length*, a proxy is substituted for the actual property list as long as the constituent objects of that property list are not being accessed. If *mutable* is YES and the object is a dictionary or an array, the re-composed object is made mutable.

NSDictionary

Inherits From: NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying
NSObject (NSObject)

Declared In: Foundation/NSDictionary.h

Class Description

The NSDictionary class declares the programmatic interface to objects that manage immutable associations of keys and values. You use this class when you need a convenient and efficient way to retrieve data associated with an arbitrary key.

A key-value pair within an NSDictionary is called an *entry.* Each entry consists of an string object that represents the key and another object (of any class) that is that key's value. You establish the entries when the NSDictionary is created, and thereafter the entries can't be modified. (The complementary class NSMutableDictionary defines objects that manage modifiable collections of entries. See the NSMutableDictionary class specification for more information.)

Internally, an NSDictionary uses a hash table to organize its storage and to provide rapid access to a value given the corresponding key. However, the methods defined for this class insulate you from the complexities of working with hash tables, hashing functions, or the hashed value of keys. These methods take key values directly, not their hashed form.

Generally, you instantiate an NSDictionary by sending one of the **dictionary...** messages to the class object. These methods return an NSDictionary containing the associations specified as arguments to the method. Each key argument is copied and the copy is added to the NSDictionary. Each corresponding value object receives a **retain** message to ensure that it won't be deallocated prematurely.

NSDictionary's three primitive methods—**count** and **objectForKey:** and **keyEnumerator**—provide the basis for all the other methods in its interface. The **count** method returns the number of entries in the object, **objectForKey:** returns the value associated with the given key, and **keyEnumerator** returns an object that lets you step through entries in the dictionary.

The other methods declared here operate by invoking one or more of these primitives. The non-primitive methods provide convenient ways of accessing multiple entries at once. The **description...** methods and the **writeToFile:atomically:** method cause an NSDictionary to generate a description of itself and store it in a string object or a file.

Exceptions

NSSet implements the encodeWithCoder: method, which raises NSInternalInconsistencyException if the number of objects enumerated for encoding turns out to be unequal to the number of objects in the set.

Creating and Initializing an NSDictionary

+ (id)**allocWithZone:**(NSZone *)*zone*

Creates and returns an uninitialized NSDictionary in *zone*.

+ (id)**dictionary**

Creates and returns an empty NSDictionary.

+ (id)**dictionaryWithContentsOfFile:**(NSString *)*path*

Creates and returns an NSDictionary from the keys and values found in the file specified by *path*.

+ (id)**dictionaryWithObjects:**(NSArray *)*objects* **forKeys:**(NSArray *)*keys*

Creates and returns an NSDictionary that associates objects from the *objects* array with keys from the *keys* array. Keys must be strings. Raises NSInvalidArgumentException if the number of *objects* is not equal to the number of *keys*.

+ (id)**dictionaryWithObjects:**(id *)*objects* **forKeys:**(id *)*keys* **count:**(unsigned int)*count*

Creates and returns an NSDictionary containing *count* objects from the *objects* array. The objects are associated with *count* keys taken from the *keys* array.

+ (id)**dictionaryWithObjectsAndKeys:**(id)*firstObject*, ...

Creates and returns an NSDictionary that associates objects and keys from the argument list. The list must be in form: *object1*, *key1*, *object2*, *key2*, ..., **nil**. Raises NSInvalidArgumentException if any of the keys are nil, or if any of the keys are not of the NSString class.

– (id)**initWithContentsOfFile:**(NSString *)*path*

Initializes a newly allocated NSDictionary using the keys and values found in *filename*.

– (id)**initWithDictionary:**(NSDictionary *)*dictionary*

Initializes a newly allocated NSDictionary by placing in it the keys and values contained in *otherDictionary.*

– (id)**initWithObjectsAndKeys:**(id)*firstObject*,...

Initializes a newly allocated NSDictionary by placing in it the objects and keys from the argument list. The list must be in form: *object1*, *key1*, *object2*, *key2*, ..., **nil**. Raises NSInvalidArgumentException if any of the keys are nil, or if any of the keys are not of the NSString class.

– (id)**initWithObjects:**(NSArray *)*objects* **forKeys:**(NSArray *)*keys*

Initializes a newly allocated NSDictionary by associating objects from the *objects* array with keys from the *keys* array. Keys must be strings. Raises NSInvalidArgumentException if the number of objects is not equal to the number of keys.

– (id)**initWithObjects:**(id *)*objects*
forKeys:(id *)*keys*
count:(unsigned)*count*

Initializes a newly allocated NSDictionary by associating *count* objects from the *objects* array with an equal number of keys from the *keys* array. Raises NSInvalidArgumentException if any of the *objects* or *keys* are **nil**.

Accessing Keys and Values

– (NSArray *)**allKeys**

Returns an NSArray containing the receiver's keys or an empty array if the receiver has no entries.

– (NSArray *)**allKeysForObject:**(id)*object*

Finds all occurrences of the value *anObject* in the receiver and returns an array with the corresponding keys.

– (NSArray *)**allValues**

Returns an NSArray containing the dictionary's values, or an empty array if the dictionary has no entries.

– (NSEnumerator *)**keyEnumerator**

Returns an NSEnumerator that lets you access each of the receiver's keys.

– (NSEnumerator *)**objectEnumerator**

Returns an NSEnumerator that lets you access each the receiver's values.

– (id)**objectForKey:**(id)*aKey*

Returns an entry's value given its key, or **nil** if no value is associated with *aKey.*

Counting Entries

– (unsigned)**count**

Returns the number of entries in the receiver.

Comparing Dictionaries

– (BOOL)**isEqualToDictionary:**(NSDictionary *)*other*

Compares the receiver to *otherDictionary.* If the contents of *otherDictionary* are equal to the contents of the receiver, this method returns YES. If not, it returns NO.

Storing Dictionaries

– (NSString *)**description**

Returns a string that represents the contents of the receiver.

– (NSString *)**descriptionInStringsFileFormat**

Returns a string that represents the contents of the receiver. Key-value pairs are represented in a appropriate for use in ".strings" files

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary*

Returns a string representation of the NSDictionary object. Included are the key and values that represent the locale data from *localeDictionary.*

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary* **indent:**(unsigned int)*level*

Returns a string representation of the NSDictionary object. Included are the key and values that represent the locale data from *localeDictionary.* Elements are indented from the left margin by *level* + 1 multiples of four spaces, to make the output more readable.

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*useAuxiliaryFile*

Writes a textual description of the contents of the receiver to *filename*. If *useAuxiliaryFile* is YES, the data is written to a backup file and then, assuming no errors occur, the backup file is renamed to the intended file name.

NSDistantObject

Inherits From: NSProxy

Conforms To: NSCoding
NSObject (NSProxy)

Declared In: Foundation/NSDistantObject.h

Class Description

The NSDistantObject class declares the programmatic interface to objects that serve as proxies to remote *real* objects.

Your application does not in general need to explicitly create NSDistantObject objects—they are created automatically when you create NSConnection objects for a remote object.

Exceptions

NSDistantObject raises an NSInternalInconsistencyException for a variety of exceptions resulting from internal consistency failures.

Building a Proxy

+ (NSDistantObject *)**proxyWithLocal:**(id)*target* **connection:**(NSConnection *)*connection*
Builds and returns a local proxy for a local object *target*, forming a remote proxy on the other side of *connection*.

+ (NSDistantObject *)**proxyWithTarget:**(id)*target* **connection:**(NSConnection *)*connection*
Builds and returns a remote proxy where *target* is an object on the other side of *connection*.

Initializing a Proxy

– (id)**initWithLocal:**(id)*target* **connection:**(NSConnection *)*connection*
Builds a local proxy for a local object *target*, forming a remote proxy on the other side of *connection*. You may not retain or otherwise use this proxy.

– (id)**initWithTarget:**(id)*target* **connection:**(NSConnection *)*connection*
Builds a remote proxy where *target* is an object on the other side of *connection*. It may deallocate and return **nil** if this target is already known on the connection. This is the designated initializer for subclasses.

Specifying a Protocol

– (void)**setProtocolForProxy:**(Protocol *)*proto* — Sets the proxy's protocol to *proto* for efficiency.

Returning the Proxy's Connection

– (NSConnection *)**connectionForProxy** — Returns the NSConnection instance used by the proxy.

NSEnumerator

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSUtilities.h

Class Description

NSEnumerator is a simple abstract class whose instances enumerate collections of other objects. Collection objects—such as NSSets, NSArrays, and NSDictionaries—provide NSEnumerator objects that can traverse their contents. You send **nextObject** repeatedly to an NSEnumerator to have it return the next object in the collection. When there are no more objects to return, **nextObject** returns **nil**.

Collection classes include methods that return an enumerator appropriate to the type of collection. NSArray has two methods that return an NSEnumerator object, **objectEnumerator** and **reverseObjectEnumerator** (the former traverses the array starting at its first object, while the latter starts with the last object and continues backward through the array to the first object). NSSet's **objectEnumerator** provides an enumerator for sets. NSDictionary has two enumerator-providing methods: **keyEnumerator** and **objectEnumerator**.

Note: Collections shouldn't be modified during enumeration. NSEnumerator imposes this restriction to improve enumeration speed.

Traversing a Collection

– (id)**nextObject**
Returns the next object in the collection being enumerated (for example, an NSArray or NSDictionary). Returns **nil** when the collection has been traversed.

NSException

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: Foundation/NSException.h

Class Description

The NSException class provides an object-oriented way for applications to announce and react to exceptional conditions.

An exceptional condition is one that interrupts the normal flow of program execution. Each application can interpret different types of conditions as exceptional. For example, one application might view as exceptional the attempt to save a file in a directory that's write-protected. In this sense, an exceptional condition can be equivalent to an error. Another application might interpret the user's keypress as an exceptional condition—an indication that a long-running process should be aborted.

Raising an Exception

Once an exceptional condition is detected, it must be propagated to the routine or routines that will handle it, a process referred to as "raising an exception." In the OpenStep exception handling system, exceptions are raised by instantiating an exception object and sending it a **raise** message.

Exception objects encapsulate:

- a name. A short NSString that is used to uniquely identify the exception
- a reason. A longer NSString that contains a "human-readable" reason for the exception. This reason object is printed when the exception object is printed using the "%@" format.
- *userInfo*. An NSDictionary object that you can use to supply application-specific data to the exception handler. For example, if a function's return value caused the exception to be raised, you could pass the return value to the exception handler through the *userInfo* dictionary. Or, if the exception handler displays a panel in response to the exception, *userInfo* could contain the text string to be displayed in the panel.

Handling an Exception

Sending a **raise** message to an exception object initiates the propagation of the exception and passes data about it. Where and how the exception is handled depends on where you send the message from. Let's first look at a simple case.

In general, a **raise** message is sent to an exception object within the domain of an exception handler. An exception handler is a control structure created by the macros NS_DURING, NS_HANDLER, and NS_ENDHANDLER.

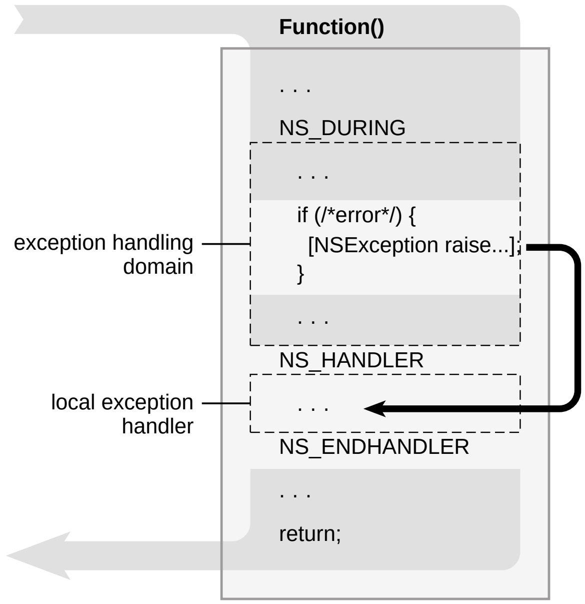

Figure 2-2. Exception Handling Domain and Handler

The section of code between NS_DURING and NS_HANDLER is the exception handling domain; the section between NS_HANDLER and NS_ENDHANDLER is the local exception handler. The normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if an exception is raised. Sending a **raise** message to an exception object causes program control to jump to the first executable line following NS_HANDLER, as indicated by the black arrow.

An exception can be raised directly within the exception handling domain, or indirectly from one of the methods or functions invoked from the domain. No matter how deeply in a call sequence an exception is raised, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in the next section). In this way, exceptions raised at a low level can be caught at a high level.

If an exception is raised and execution begins within the local exception handler, it either continues until all appropriate statements are executed or the exception is raised again to invoke the services of an encompassing exception handler, as described in the next section.

If the exception isn't raised again, execution within the local exception handler continues until it leaves the local handler by:

- "Falling off the end"
- Calling NS_VALUERETURN()
- Calling NS_VOIDRETURN

Note: A simple return from the exception-handling domain is not permitted.

"Falling off the end" is simply the normal execution pathway introduced above. After all appropriate statements within the domain are executed (and no exception is raised), execution continues on the line following NS_ENDHANDLER. Alternatively, you can return control to the caller from within the domain by calling NS_VALUERETURN() or NS_VOIDRETURN, depending on whether you need to return a value.

You can't use **goto** or **return()** to exit an exception handling domain—errors will result. Nor can you use **setjmp()** and **longjmp()** if the jump entails crossing an NS_DURING statement. Since in many cases you won't know if the code that your program calls has exception handling domains within it, it's generally not recommended that you use **setjmp()** and **longjmp()** in your application.

Nested Exception Handlers

Exception handlers can be nested so that an exception raised in an inner domain can be treated by the local exception handler and any number of encompassing exception handlers. The following diagram illustrates the use of nested exception handlers, and is discussed in the text that follows.

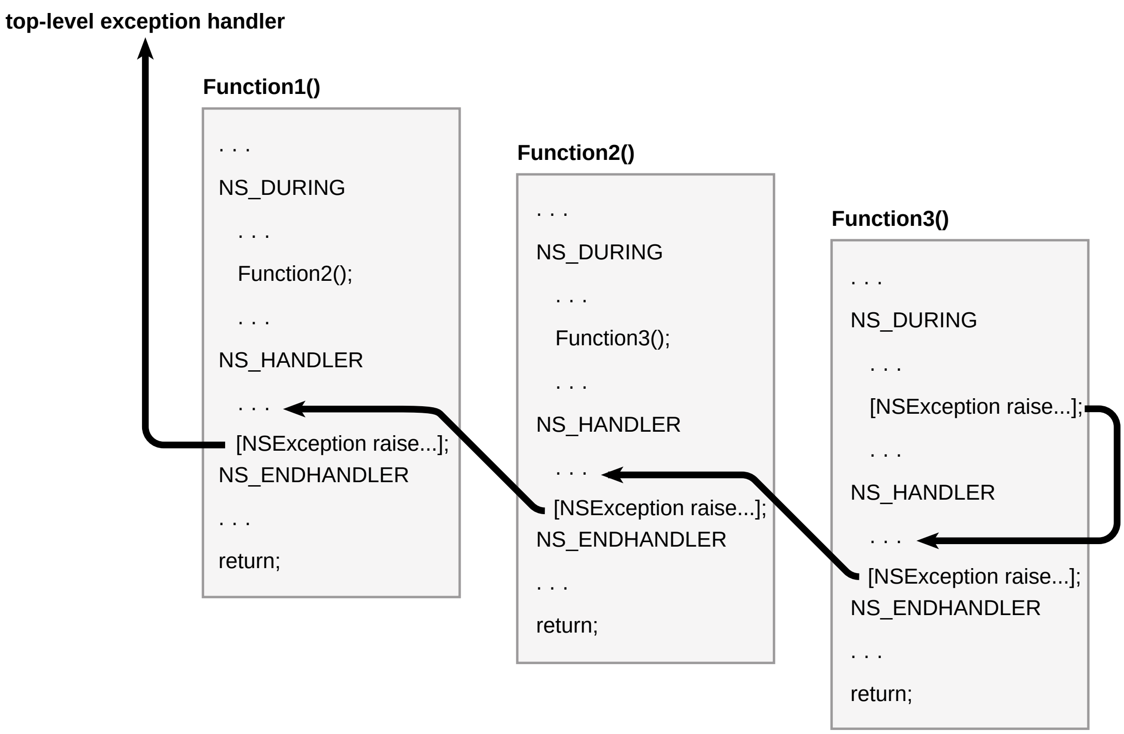

Figure 2-3. Nested Exception Handlers

An exception raised within Function3's domain causes execution to jump to its local exception handler. In a typical application, this exception handler checks the values contained the NSException object to determine the nature of the exception. For exception types that it recognizes, the local handler responds and then sends a **raise** message to the exception object to pass notification of the exception to the handler above it (in this case, the handler in

Function2). Function2's exception handler does the same and then raises the exception to Function1's handler. Finally, Function1's handler re-raises the exception. Since there's no exception handling domain above Function1, the exception is transferred to a default top-level error handler. For applications based on the Application Kit, this top-level handler invokes NSApplication's **reportException:** method, which writes an error message to the console.

An exception that's re-raised appears to the next higher handler just as if the initial exception had been raised within its own exception handling domain.

Raising an Exception Outside of an Exception Handler

If an exception is raised outside of any exception handler, it's intercepted by the uncaught exception handler, a function set by **NSSetUncaughtExceptionHandler()** and returned by **NSUncaughtExceptionHandler()**. You can change the way uncaught exceptions are handled by using **NSSetUncaughtExceptionHandler()** to establish a different procedure as the handler. However, because of the design of the Application Kit, it's rare for an exception to be raised outside of an exception handling domain. The NSApplication object's event loop itself is within an exception handling domain. On each cycle of the loop, the NSApplication object retrieves an event and sends an event message to the appropriate object in the application. Thus, the code you write for custom objects (as well as the code for Application Kit objects) is executed within the context of the event loop's exception handler.

Predefined Exceptions

OpenStep predefines a number of exception names. These exceptions are listed in **NSException.h**; for example:

```
extern NSString *NSGenericException;
extern NSString *NSRangeException;
extern NSString *NSInvalidArgumentException;
```

For a complete list of global exception names, see the "Types and Constants" sections of this manual. You can catch any of these exceptions from within your exception handler by comparing the exception's name with these predefined exception names.

Creating and Raising Exceptions

+ (NSException *)**exceptionWithName:**(NSString *)*name*
reason:(NSString *)*reason*
userInfo:(NSDictionary *)*userInfo*

Creates an exception object, assigning it *name* as its name, *reason* as its human-readable explanation, and *userInfo* as arbitrary data that will accompany the exception.

+ (volatile void)**raise:**(NSString *)*name*
format:(NSString *)*format*,...

Creates and raises an exception with name *name* and a reason constructed from *format* and the following arguments in the manner of **printf()**. The user-defined information is **nil**. Invokes **raise** as part of its implementation.

+ (volatile void)**raise:**(NSString *)*name*
format:(NSString *)*format*
arguments:(va_list)*argList*

Creates and raises an exception with name *name* and a reason constructed from *format* and the arguments in *argList*, in the manner of **vprintf()**. The user-defined information is **nil**. Invokes **raise** as part of its implementation.

– (id)**initWithName:**(NSString *)*name*
reason:(NSString *)*reason*
userInfo:(NSDictionary *)*userInfo*

Initializes a newly allocated exception object, assigning it *name* as its name, *reason* as its human-readable explanation, and *userInfo* as arbitrary data that will accompany the exception.

– (volatile void)**raise**

Raises the exception, causing program flow to jump to the enclosing error handler.

Querying Exceptions

– (NSString *)**name**

Returns the exception's name. See **exceptionWithName:reason:userInfo:**.

– (NSString *)**reason**

Returns the exception's reason. See **exceptionWithName:reason:userInfo:**.

– (NSDictionary *)**userInfo**

Returns the exception's user-defined data. See **exceptionWithName:reason:userInfo:**.

NSInvocation

Inherits From: NSObject

Conforms To: NSCoding
NSObject (NSObject)

Declared In: Foundation/NSInvocation.h

Class Description

Objects of the NSInvocation class provide a system-independent means to construct message calls to other objects. An NSInvocation object constructs a *target* object to which a message can be sent, a *selector* for that method, an *argument list* for the selector, and a return value. NSInvocation objects provide great flexibility in that the methods, method arguments, and targets of the methods may be constructed dynamically.

The final sending of the message to the target object can be performed at any time, independent of constructing the invocation. For example, methods could be dispatched based on timer events. In addition, return values from the methods are stored in the NSInvocation object and can be retrieved at any later stage in processing.

Also see NSMethodSignature for a description of how to construct method signatures.

The **Foundation/NSInvocation.h** header file defines two macros that may be used as constructors for invocations:

NSInvocation *invocation = NS_MESSAGE(*target***,** *message***)**
builds an invocation containing a *message* to a known *target* object. *target* is an object id. *message* consists of a selector followed by any arguments, just like an Objective-C message.

NSInvocation *invocation = NS_INVOCATION(*class***,** *message***)**
builds an invocation containing a *message* to the untargeted class object *class*. *message* consists of a selector followed by any arguments, just like an Objective-C message.

Creating Invocations

+ (NSInvocation *)**invocationWithMethodSignature:**(NSMethodSignature *)*sig*
Returns an invocation object able to construct calls to objects using method selectors with type signatures described by *sig*. Raises NSInvalidArgumentException if *sig* is **nil**.

Managing Invocation Arguments

– (BOOL)**argumentsRetained**

Returns YES if arguments are retained.

– (void)**getArgument:**(void *)*argumentLocation* **atIndex:**(int)*index*

Copies the argument stored at *index* into the storage pointed to by *argumentLocation* where 2 is the index of the first argument, 3 is the index of the second, and so on.

– (void)**getReturnValue:**(void *)*retLoc*

Copies the invocation's return value into the storage pointed to by *retLoc*.

– (NSMethodSignature *)**methodSignature**

Returns the invocation's method signature object.

– (void)**retainArguments**

By default, target and arguments are not retained, and C strings are not copied. This method instructs the invocation to retain its arguments, target, and make copies of C strings. This method is invoked automatically by timers. This method should be invoked whenever the dynamic scope of the invocation can exceed its arguments.

– (SEL)**selector**

Returns the invocation's selector.

– (void)**setArgument:**(void *)*argumentLocation* **atIndex:**(int)*index*

Sets the argument stored at *index* to the storage pointed to by *argumentLocation* where 2 is the index of the first argument, 3 is the index of the second, and so on..

– (void)**setReturnValue:**(void *)*retLoc*

Sets the invocation's return value to that indicated by *retLoc*.

– (void)**setSelector:**(SEL)*selector*

Sets the invocation's selector to *selector.*

– (void)**setTarget:**(id)*target*

Sets the invocation's target to *target*.

– (id)**target**

Returns the invocation's target; returns **nil** if there is no target.

Dispatching an Invocation

– (void)**invoke**

Causes the message encoded in the invocation to be dispatched to its target.

– (void)**invokeWithTarget:**(id)*target*

Causes the message encoded in the invocation to be dispatched to *target*.

NSLock

Inherits From: NSObject

Conforms To: NSLocking
NSObject (NSObject)

Declared In: Foundation/NSLock.h

Class Description

An NSLock is used to protect critical regions of code. A lock is created once and is subsequently used to protect one or more regions of code. If a region of code is in use, an NSLock waits using the **condition_wait()** function, so the thread doesn't busy-wait. The following example shows the use of an NSLock with the methods **lock** and **unlock** defined in the NSLocking protocol:

```
NSLock  *theLock = [NSLock new];       // done once!
/* ... other code */
[theLock lock];
/* ... possibly a long time of fussing with global data... */
[theLock unlock];
```

The NSConditionLock, NSLock, and NSRecursiveLock classes all implement the NSLocking protocol with various features and performance characteristics; see the other class descriptions for more information.

Acquiring a Lock

– (BOOL)**tryLock** Attempts to acquire a lock. Returns YES if successful and NO otherwise. Returns immediately.

NSMethodSignature

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSMethodSignature.h

Class Description

NSMethodSignature provides the programmatic interface to objects that provide access to the "type signatures" of an object's methods—that is, the types of the arguments and return value. A *method signature* is used by the distributed objects machinery to determine how to correctly encode method names and arguments for the underlying inter-process communications. The typical use of method signatures is when a message is sent to a remote object *via* a proxy. If the proxy doesn't know the types of arguments a remote object will use, the proxy first has to query the remote object for its method signature object, which specifies the types the method requires as arguments. The proxy then knows how to encode the data it has been passed and forward it correctly to the real object.

You create a method signature object by sending a **signatureWithObjCTypes** method to the NSMethodSignature class object, passing a "C"-style character string which specifies the method's return types and argument types.

Given a method signature, all other available instance methods query the object for information about the signature, such as its return type, number of arguments, stack frame size (obviously architecture-dependent), and so on.

Also see NSInvocation for the class which can use method signature objects to send messages to other objects.

Creating a Method Signature

+ (NSMethodSignature *)**signatureWithObjCTypes:**(const char *)*types*

Creates a method signature object given *types*, a string encoding the method return and argument types.

Querying a Method Signature

– (NSArgumentInfo)**argumentInfoAtIndex:**(unsigned)*index*

Returns information about the argument at *index*. Indices begin with 0. The "hidden" arguments **self** and **_cmd** are indexed at 0 and 1; method-specific arguments begin at index 2. If *index* is too large for the actual number of arguments, NSInvalidArgumentException is raised.

– (unsigned)**frameLength**

Returns the number of bytes that the arguments, taken together, would occupy on the stack.

– (BOOL)**isOneway** Returns YES if the method is asynchronous (that is, it returns without waiting for the receiver to finish processing it), and NO otherwise.

– (unsigned)**methodReturnLength** Returns the number of bytes required by the return value.

– (char *)**methodReturnType** Returns a string encoding the return type of the method. (What the characters in the string represent is usually defined by some implementation-dependent runtime types.)

– (unsigned)**numberOfArguments** Returns the number of arguments recorded in the receiver. This will be at least two, since it includes the "hidden" arguments, **self** and **_cmd**, which are the first two arguments passed to every method implementation.

NSMutableArray

Inherits From: NSArray : NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying (NSArray)
NSObject (NSObject)

Declared In: Foundation/NSArray.h

Class Description

The NSMutableArray class declares the programmatic interface to objects that manage a modifiable array of objects. This class adds insertion and deletion operations to the basic array-handling behavior it inherits from NSArray.

The array operations that NSMutableArray declares are conceptually based on these three methods:

addObject:
replaceObjectAtIndex:withObject:
removeLastObject

The other methods in its interface provide convenient ways of inserting an object into a specific slot in the array and of removing an object based on its identity or position in the array.

When an object is removed from a mutable array it receives a **release** message, which can cause it to be deallocated. Note that if your program keeps a reference to such an object, the reference may become invalid unless you remember to send the object a **retain** message before it's removed from the array. For example, the third statement below could result in a run-time error, except for the **retain** message in the first statement:

```
id anObject = [[anArray objectAtIndex:0] retain];
[anArray removeObjectAtIndex:0];
[anObject someMessage];
```

Implementing Subclasses of NSMutableArray

Although conceptually the interface to the NSMutableArray class is based on the three methods listed above, for performance reasons two others—**insertObject:atIndex:** and **removeObjectAtIndex:**—also directly access the object's data. These two methods could be implemented using the methods listed above but in doing so would incur unnecessary overhead from the **retain** and **release** messages that objects would receive as they are shifted to accommodate the insertion or deletion of an element. Thus, if you create a subclass of NSMutableArray, you should override all five primitive methods so that the other methods in NSMutableArray's interface work properly.

Creating and Initializing an NSMutableArray

+ (id)**allocWithZone:**(NSZone *)*zone* — Creates and returns an uninitialized NSMutableArray in *zone.*

+ (id)**arrayWithCapacity:**(unsigned int)*aNumItems* — Creates and returns an NSMutableArray, giving it enough allocated memory to hold *numItems* objects.

– (id)**initWithCapacity:**(unsigned int)*aNumItems* — Initializes a newly allocated NSMutableArray, giving it enough memory to hold *numItems* objects.

Adding Objects

– (void)**addObject:**(id)*anObject* — Inserts *anObject* at the end of the array. Raises NSInvalidArgumentException if *anObject* is **nil**.

– (void)**addObjectsFromArray:**(NSArray *)*anotherArray*

Adds the objects contained in *anotherArray* to the end of the receiver's array.

– (void)**insertObject:**(id)*anObject* **atIndex:**(unsigned int)*index*

Inserts anObject into the array at *index*. Raises NSInvalidArgumentException if *anObject* is **nil**. Raises NSRangeException if *index* is outside of the bounds of the array.

Removing Objects

– (void)**removeAllObjects** — Empties the array of all its elements.

– (void)**removeLastObject** — Removes the last object in the array and sends it a **release** message. Raises NSRangeException if there are no objects in the array.

– (void)**removeObject:**(id)*anObject* — Removes all occurrences of *anObject*. **isEqual:** is used to test for *anObject*.

– (void)**removeObjectAtIndex:**(unsigned int)*index* — Removes the object at *index* and moves all elements beyond *index* up one slot to fill the gap. Raises NSRangeException if *index* is outside of the bounds of the array.

– (void)**removeObjectIdenticalTo:**(id)*anObject* — Removes all elements having the same **id** as *anObject*.

– (void)**removeObjectsFromIndices:**(unsigned int*)*indices*
numIndices:(unsigned int)*count* — Removes objects at the positions specified in the *indices* array, which has *count* elements. Raises NSRangeException if any of the *indices* is outside of the bounds of the array. This method is provided for efficiency reasons; it will not work if the receiver is a proxy to an array in another process.

– (void)**removeObjectsInArray:**(NSArray *)*otherArray*

Removes from the receiver the objects found in *otherArray.*

Replacing Objects

– (void)**replaceObjectAtIndex:**(unsigned int)*index* **withObject:**(id)*anObject*

Replaces the object at *index* with *anObject*. Raises NSInvalidArgumentException if *anObject* is **nil**. Raises NSRangeException if *index* is not within the bounds of the array.

– (void)**setArray:**(NSArray *)*otherArray*

Sets the contents of the receiver to the elements in *otherArray*

Sorting Elements

– (void)**sortUsingFunction:**(int (*)(id *element1*, id *element2*,void **userData*))*comparator* **context:**(void *)*context*

Sorts the receiver's elements in ascending order as defined by the comparison function *comparator. context* is passed as the function's third argument.

– (void)**sortUsingSelector:**(SEL)*comparator*

Sorts the receiver's elements in ascending order as defined by the comparison method *comparator.*

NSMutableCharacterSet

Inherits From: NSCharacterSet : NSObject

Conforms To: NSCopying, NSMutableCopying
NSCoding, NSCopying, NSMutableCopying (NSCharacterSet)
NSObject (NSObject)

Declared In: Foundation/NSCharacterSet.h

Class Description

The NSMutableCharacterSet class declares the programmatic interface to objects that construct mutable *descriptions* of character sets in the Unicode character encoding. Ha ving constructed such character set descriptions using methods described in the NSCharacterSet class, you can use the methods described here to modify the character sets dynamically.

Adding and Removing Characters

– (void)**addCharactersInRange:**(NSRange)*aRange* Adds to the receiver the Unicode characters whose values are given by *aRange*.

– (void)**addCharactersInString:**(NSString *)*aString* Adds the characters in *aString* to those in the receiver.

– (void)**removeCharactersInRange:**(NSRange)*aRange*
Removes from the receiver the Unicode characters whose values are given by *aRange*.

– (void)**removeCharactersInString:**(NSString *)*aString*
Removes from the receiver the characters in *aString*.

Combining Character Sets

– (void)**formIntersectionWithCharacterSet:**(NSCharacterSet *)*otherSet*
Modifies the receiver so that it contains only those characters that exist in both the receiver and in *otherSet*.

– (void)**formUnionWithCharacterSet:**(NSCharacterSet *)*otherSet*
Modifies the receiver so that it contains all characters that exist in either the receiver or *otherSet*, barring duplicates.

Inverting a Character Set

– (void)**invert**

Replaces all of the characters in the receiver with all the characters it didn’t previously contain.

NSMutableData

Inherits From: NSData : NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying (NSData)
NSObject (NSObject)

Declared In: Foundation/NSData.h
Foundation/NSSerialization.h

Class Description

The NSMutableData class declares the programmatic interface to objects that contain modifiable data in the form of bytes. This class inherits all read-only access methods from its superclass, NSData, and declares only those methods that permit the modification of the data.

NSMutableData's two primitive methods—**mutableBytes** and **setLength:**—provide the basis for all the other methods in its interface. The **mutableBytes** method returns a pointer for writing into the bytes contained in the mutable data object. **setLength:** allows you to truncate or extend the length of a mutable data object.

The **appendBytes:length:** and **appendData:** methods let you append bytes or the contents of another data object to a mutable data object. You can replace a range of bytes in a mutable data object with either zeroes (using the **resetBytesInRange:** method), or with different bytes (using the **replaceBytesInRange:withBytes:** method).

This class declares various serialization methods that enable architecture-independent serialization of arbitrary Objective C types.

Creating an NSMutableData Object

+ (id)**allocWithZone:**(NSZone *)*zone* — Creates and returns an uninitialized mutable data object from *zone*.

+ (id)**dataWithCapacity:**(unsigned int)*numBytes* — Creates and returns a mutable data object, initially allocating enough memory to hold *numBytes* bytes.

+ (id)**dataWithLength:**(unsigned int)*length* — Creates and returns a mutable data object, giving it enough memory to hold *length* bytes. Fills the object with zeroes up to *length*.

– (id)**initWithCapacity:**(unsigned int)*capacity* — Initializes a newly allocated mutable data object, giving it enough memory to hold *capacity* bytes. Sets the length of the data object to 0.

– (id)**initWithLength:**(unsigned int)*length* — Initializes a newly allocated mutable data object, giving it enough memory to hold *length* bytes. Fills the object with zeroes up to *length*.

Adjusting Capacity

– (void)**increaseLengthBy:**(unsigned int)*extraLength*
Increases the length of a mutable data object by *extraLength* zero-filled bytes.

– (void *)**mutableBytes**
Returns a pointer to the bytes in a mutable data object, enabling you to modify the bytes.

– (void)**setLength:**(unsigned int)*length*
Extends or truncates the length of a mutable data object by *length* bytes. If the mutable data object is extended, the additional bytes are zero-filled.

Appending Data

– (void)**appendBytes:**(const void *)*bytes*
length:(unsigned int)*length*
Appends *length* bytes to a mutable data object from the buffer *bytes*.

– (void)**appendData:**(NSData *)*other*
Appends the contents of the data object *other* to the receiver.

Modifying Data

– (void)**replaceBytesInRange:**(NSRange)*aRange*
withBytes:(const void *)*bytes*
Replaces the receiver's bytes located in *aRange* with *bytes*. Raises an NSRangeException if *aRange* is not within the range of the receiver's data.

– (void)**resetBytesInRange:**(NSRange)*aRange*
Replaces the receiver's bytes located in *aRange* with zeros. Raises an NSRangeException if *aRange* is not within the range of the receiver's data.

Serializing Data

– (void)**serializeAlignedBytesLength:**(unsigned int)*length*
Prepares bytes for an **appendBytes:length:** invocation by serializing them. If the *length* of the bytes will cause extension past the page size, this method encodes header information, creating a hole so that all bytes in the data object are aligned on page boundaries.

– (void)**serializeDataAt:**(const void *)*data*
ofObjCType:(const char *)*type*
context:
(id <NSObjCTypeSerializationCallBack>)*callback*
Serializes whatever data element is referenced by *data*, interpreting it by the Objective C type specifier *type*. If the data element is an object other than an instance of NSDictionary, NSArray, NSString, or NSData, further definition of the object can occur through a callback from object *callback*. All Objective C types are currently supported except **unions** and **void *.** Pointers refer to a single item.

– (void)**serializeInt:**(int)*value*

Serializes the integer *value* by encoding it as a character representation.

– (void)**serializeInt:**(int)*value* **atIndex:**(unsigned int)*index*

Serializes the integer *value* by encoding it as a character representation and replaces the encoded value at the specified *index* in the data.

– (void)**serializeInts:**(int *)*intBuffer* **count:**(unsigned int)*numInts*

Serializes *numInts* count of integers in *intBuffer* by encoding each integer as a character representation.

– (void)**serializeInts:**(int *)*intBuffer* **count:**(unsigned int)*numInts* **atIndex:**(unsigned int)*index*

Serializes *numInts* count of integers in *intBuffer* by encoding each integer, starting at the specified *index*, and replacing each corresponding integer encoding serially.

NSMutableDictionary

Inherits From: NSDictionary : NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying (NSDictionary)
NSObject (NSObject)

Declared In: Foundation/NSDictionary.h

Class Description

The NSMutableDictionary class declares the programmatic interface to objects that manage mutable associations of keys and values. With its two efficient primitive methods—**setObject:forKey:** and **removeObject:forKey:**—this class adds modification operations to the basic operations it inherits from NSDictionary.

The other methods declared here operate by invoking one or both of these primitives. The derived methods provide convenient ways of adding or removing multiple entries at a time.

When an entry is removed from a mutable dictionary, the key and value objects that make up the entry receive a **release** message, which can cause them to be deallocated. Note that if your program keeps a reference to such objects, the reference will become invalid unless you remember to send the object a **retain** message before it's removed from the dictionary. For example, the third statement below could result in a run-time error, except for the **retain** message in the first statement:

```
id anObject = [[aDictionary objectForKey:theKey] retain];
[aDictionary removeObjectForKey:theKey];
[anObject someMessage];
```

Allocating and Initializing

+ (id)**allocWithZone:**(NSZone *)*zone* — Creates and returns an uninitialized NSMutableDictionary in *zone*.

+ (id)**dictionaryWithCapacity:**(unsigned int)*aNumItems* — Creates and returns an NSMutableDictionary, giving it enough allocated memory to hold *numEntries* entries.

– (id)**initWithCapacity:**(unsigned int)*aNumItems* — Initializes a newly allocated NSMutableDictionary, giving it enough allocated memory to hold *numEntries* entries.

Adding and Removing Entries

– (void)**addEntriesFromDictionary:**(NSDictionary *)*otherDictionary*
Adds the entries from *otherDictionary* to the receiver.

– (void)**removeAllObjects**
Empties the receiver of its entries.

– (void)**removeObjectForKey:**(id)*theKey*
Removes *theKey* and its associated value object from the dictionary. Raises NSInvalidArgumentException if *aKey* is **nil**.

– (void)**removeObjectsForKeys:**(NSArray *)*keyArray*
Removes from the receiver one or more entries as identified by the keys in *keyArray.*

– (void)**setObject:**(id)*anObject* **forKey:**(id)*aKey*
Adds an entry to the receiver, consisting of *anObject* and its corresponding key *aKey.* Raises NSInvalidArgumentException if either *anObject* or *aKey* is **nil.**

– (void)**setDictionary:**(NSDictionary *)*otherDictionary*
Sets the contents of the receiver to the keys and values in *other.*

NSMutableSet

Inherits From: NSSet : NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying (NSSet)
NSObject (NSObject)

Declared In: Foundation/NSSet.h

Class Description

The NSMutableSet class declares the programmatic interface to an object that manages a mutable set of objects. NSMutableSet provides support for the mathematical concept of a *set*. A set, both in its mathematical sense, and in the OpenStep implementation of NSMutableSet, is an *unordered* collection of distinct elements. OpenStep also provides the NSCountedSet class for a mutable set that can contain multiple instances of the same element, and provides the NSSet class for creating and managing immutable sets. In general, you should use NSSet unless you really need a mutable set.

Use set objects as an alternative to array objects when the order of elements is not important, but performance in testing whether an object is contained in the set *is* a consideration—while arrays are ordered, testing for membership is slower than with sets.

Objects in a set must respond to **hash** and **isEqual:** methods. See the NSObject protocol for details on **hash** and **isEqual:**.

Generally, you instantiate an NSMutableSet object by sending one of the **set…** methods to the NSMutableSet class object, as described in the method descriptions for NSSet. These methods return an NSMutableSet object containing the elements (if any) you pass in as arguments. Newly created instances of NSMutableSet created by invoking the **set** method can be populated with objects using any of the **init…** methods. **initWithObjects::** is the designated initializer for this class.

Objects are added to an NSMutableSet using **addObject:**, which adds a single specified object to the set, **addObjectsFromArray:**, which adds all objects from a specified array to the set, or by **unionSet:**, which adds all the objects from another set to this set.

Objects are removed from an NSMutableSet using any of the methods **intersectSet:**, **minusSet:**, **removeAllObjects**, or **removeObject:**.

Allocating and Initializing an NSMutableSet

+ (id)**allocWithZone:**(NSZone *)*zone* — Creates and returns an uninitialized set object in *zone.*

+ (id)**setWithCapacity:**(unsigned)*numItems* — Creates and returns a set object, giving it enough allocated memory to hold *numItems* objects.

– (id)**initWithCapacity:**(unsigned)*numItems* — Initializes a newly allocated set object, giving it enough allocated memory to hold *numItems* objects.

Adding Objects

– (void)**addObject:**(id)*object* — Adds *object* to the set, unless *object* is equal to some object already in the set.

– (void)**addObjectsFromArray:**(NSArray *)*array* — Adds to the set all the objects in *array*, by calling **addObject:** for each one.

– (void)**unionSet:**(NSSet *)*other* — Adds to the receiving set all the objects in *other*, by calling **addObject:** for each one.

Removing Objects

– (void)**intersectSet:**(NSSet *)*other* — Removes from the receiving set every object that's not equal to any object in *other*, by calling **removeObject:** for each one.

– (void)**minusSet:**(NSSet *)*other* — Removes from the receiving set every object that's equal to some object in *other*, by calling **removeObject:** for each one.

– (void)**removeAllObjects** — Empties the set of all its elements. (This method doesn't call **removeObject:**.)

– (void)**removeObject:**(id)*object* — If any member of the receiving set is equal to *object*, this method removes that object from the set.

NSMutableString

Inherits From: NSString : NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying (NSString)
NSObject (NSObject)

Declared In: Foundation/NSString.h

Class Description

NSMutableString (and NSString) declare the programmatic interface for objects that create and managemutable *representation-independent* character strings. For a more general overview of string classes, see the description of NSString.

NSMutableString (and NSString) are abstract classes for string manipulation. NSMutableString declares the interface to objects that inherit all the capabilities of NSString objects, but in addition allow for modification of the string data. NSString and NSMutableString provide factory methods that return autoreleased instances of unspecified subclasses of strings.

You can instantiate an NSMutableString object by sending any of the **stringWith**… methods to the NSMutableString class object. This set of methods also includes **localizedStringWithFormat:**. A newly allocated NSMutableString object can also be initialized using the **initWithCapacity**: method, to set the string to a specified capacity.

Creating Temporary Strings

+ (NSMutableString *)**localizedStringWithFormat:**(NSString *)*format,...*

Returns a string created by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string. The user's default locale is used for format information.

+ (NSMutableString *)**stringWithCString:**(const char *)*zeroTerminatedBytes*

Returns a mutable string containing the characters in *zeroTerminatedBytes*, which must be null-terminated. The *zeroTerminatedBytes* string should contain bytes in the default C string encoding.

+ (NSMutableString *)**stringWithCString:**(const char *)*bytes*
length:(unsigned int)*length*

Returns a mutable string containing *length* characters made from *bytes*. This method doesn't stop at a null byte. *bytes* should contain bytes in the default C string encoding.

+ (NSMutableString *)**stringWithCapacity:**(unsigned int)*capacity*

Returns an empty mutable string, using *capacity* as a hint for how much initial storage to reserve.

+ (NSMutableString *)**stringWithCharacters:**(const unichar *)*characters* **length:**(unsigned int)*length*

Returns a mutable string containing *characters*. The first *length* characters are copied into the string. This method doesn't stop at a null character.

+ (NSMutableString *)**stringWithContentsOfFile:**(NSString *)*path*

Returns a string containing the contents of the file specified by *path*. This method attempts to determine the encoding for the file. The string is assumed to be in Unicode encoding, but if the encoding is determined not to be Unicode, the default C string encoding is used instead.

+ (NSMutableString *)**stringWithFormat:**(NSString *)*format,...*

Returns a mutable string created by using *format* as a **printf()** style format string, and the subsequent arguments as values to be substituted into the format string.

Initializing a Mutable String

– **initWithCapacity:**(unsigned int)*capacity*

Initializes a newly allocated mutable string object, giving it enough allocated memory to hold *capacity* characters.

Modifying a String

– (void)**appendFormat:**(NSString *)*format,...*

Adds a constructed string to the receiver. The new characters are created by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string. Invokes **replaceCharactersInRange:withString:** as part of its implementation.

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

Adds the characters of *aString* to end of the receiver. Invokes **replaceCharactersInRange:withString:** as part of its implementation.

– (void)**deleteCharactersInRange:**(NSRange)*range*

Removes from the receiver the characters in *range*. This method raises an NSStringBoundsError exception if any part of *range* lies beyond the end of the string. Invokes **replaceCharactersInRange:withString:** as part of its implementation.

– (void)**insertString:**(NSString *)*aString* **atIndex:**(unsigned)*index*

Inserts the characters of *aString* into the receiver, such that the new characters begin at *index* and the existing character from *index* to the end are shifted by the length of *aString*. This method raises an NSStringBoundsError exception if *index* lies beyond the end of the string. Invokes **replaceCharactersInRange:withString:** as part of its implementation.

– (void)**replaceCharactersInRange:**(NSRange)*aRange* **withString:**(NSString *)*aString*

Inserts the characters of *aString* into the receiver, such that they replace the characters in *aRange*. This method raises an NSStringBoundsError exception if any part of *aRange* lies beyond the end of the string.

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

Replaces the characters of the receiver with those in *aString*.

NSNotification

Inherits From: NSObject

Conforms To: NSCopying
NSObject (NSObject)

Declared In: Foundation/NSNotification.h

Class Description

NSNotification objects provide a flexible way to transmit event information between objects.

Message passing—invoking a method—is the standard way to convey information between objects. However, this requires the object sending the message to know who the receiver is. At times this explicit binding of two objects is undesirable—most notably because it would tie two otherwise independent subsystems. For these instances, a looser broadcast model is introduced: An object posts a notification, which is dispatched to the appropriate receivers through a notification center.

An object may post an NSNotification object (referred to as a *notification object* or simply, a *notification*), which contains information about an object: the notification's name, its sender, and an optional dictionary containing other information. Other objects can register themselves as observers to receive notification objects when they are posted. When the event happens, the registered objects receive notifications about it. The object posting the NSNotification object, the object the notification is about, and the observer of the notification may all be different objects.

An NSNotificationCenter object registers observers for events and notifies the observers if these events occur. An object may ask an NSNotificationCenter object (also known as a *notification center*) to observe an event regarding another object. If the event occurs, the posting object tells the notification center to notify its observers that this condition has occurred. The notification center then sends a notification to all observing objects. (See the class specification of NSNotificationCenter for more on posting notification objects.)

This notification model frees an object from concern about what objects may want to observe it. An object involved with an event—or another object—may simply post a notification about that event without knowing what objects—if any—are observing the event. The notification center takes care of distributing notifications to registered observers. Another benefit of this model is to allow multiple objects to listen for notifications, an effect that might otherwise require explicitly setting up an array.

You instantiate a notification object directly by sending the **notificationWithName:object:** or **notificationWithName:object:userInfo:** messages to the NSNotification class object. You can also create notifications indirectly through the NSNotificationCenter class using the **postNotificationName:object:** and **postNotificationName:object:userInfo:** convenience methods.

You can subclass NSNotification to contain information in addition to the notification name, sender, and dictionary.

NSNotification objects are immutable objects.

The NSNotification class adopts the NSCopying protocol, making it possible to treat notifications as context-independent values that can be copied and reused. You can put notifications in an array and send the **copy** message to that array, which recursively copies every item. This essentially allows clients to deal with notifications as first class values that can be copied by collections.

Creating Notification Objects

+ (NSNotification *)**notificationWithName:**(NSString *)*aName*
object:(id)*anObject*

Returns a notification object that associates the name *aName* with the object *anObject*.

+ (NSNotification *)**notificationWithName:**(NSString *)*aName*
object:(id)*anObject*
userInfo:(NSDictionary *)*userInfo*

Returns a notification object that associates the name *aName* with the object *anObject* and the dictionary of arbitrary data *userInfo*. *userInfo* may be **nil**.

Querying a Notification Object

– (NSString *)**name**

Returns the name of the notification.

– (id)**object**

Returns the object (such as the sender) that's associated with this notification.

– (NSDictionary *)**userInfo**

Returns a dictionary object associated with this notification. Returns **nil** if there is no such object.

NSNotificationCenter

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSNotification.h

Class Description

An NSNotificationCenter object (or simply, *notification center*) is essentially a notification dispatch table. It notifies all observers of events meeting specific criteria of notification and sender. This event information is encapsulated in NSNotification objects, also known as *notification objects*, or simply, *notifications*. Client objects register themselves as observers of a specific notification originating in another object. When the condition occurs to signal a notification, some object (which may or may not be the object observed) posts an appropriate notification object to the notification center. (See the class specification of NSNotification for more on notification objects.) The notification center dispatches a message to each observer (using the selector provided by the observer), with the notification as the sole argument.

An object registers itself to observe notifications by the **addObserver:selector:name:object:** method, specifying the object and associated notification it wants to see. However, the observer need not specify both of these parameters. If it specifies only the object, it will see *all* notifications associated with that object. If the object specifies only a notification name to observe, it will see that notification for *any* object whenever it's posted.

The methods **postNotificationName:object:** and **postNotificationName:object:userInfo:** are provided as convenience methods, which both create and post notifications.

Each task has a default notification center.

As an example of using the notification center, suppose your program can perform a number of conversions on text (for instance, MIF to RTF or RTF to ASCII). You have defined a class of objects that perform those conversions, Convertor. Convertor objects might be added or removed during program execution. Your program has a client object that wants to be notified when convertors are added or removed, allowing the application to reflect the available options in a pop-up list. The client object would register itself as an observer by sending the following messages to the notification center:

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(objectAddedToConvertorList:)
    name:@"NSConverterAdded" object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(objectRemovedFromConvertorList:)
    name:@"NSConverterRemoved" object:nil];
```

When a user installs or removes a converter, the Convertor sends one of the following messages to the notification center:

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"NSConverterAdded" object:self];
```

or

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"NSConverterRemoved" object:self];
```

The notification center identifies all observers who are interested in the “NSConverterAdded” or “NSConverterRemoved” notifications by invoking the method they specified in the selector argument of **addObserver:selector:name:object:**. In the case of our example observer, the selectors are **objectAddedToConvertorList:** and **objectRemovedFromConvertorList:**. Assume the Convertor class has an instance method **convertorName** that returns the name of the Convertor object. Then the **objectAddedToConvertorList:** method might have the following implementation:

```
- (void)objectAddedToConvertorList:(NSNotification *)notification
{
    Convertor *addedConvertor = [notification object];

    // Add this to our popup (it will only be added if not there)...
    [myPopUpButton addItem:[addedConvertor convertorName]];
}
```

The convertors don’t need to know anything about the pop-up list or any other aspect of the user interface to your program.

Accessing the Default Notification Center

+ (NSNotificationCenter *)**defaultCenter**	Returns the default notification center object; used for generic notifications.

Adding and Removing Observers

– (void)**addObserver:**(id)*anObserver* **selector:**(SEL)*aSelector* **name:**(NSString *)*aName* **object:**(id)*anObject*	Registers *anObserver* and *aSelector* with the receiver so that *anObserver* receives an *aSelector* message when a notification of name *aName* is posted to the notification center by *anObject*. If *anObject* is **nil**, observer will get posted whatever the object is. If *aName* is **nil**, observer will get posted for all notifications that match *anObject*.
– (void)**removeObserver:**(id)*anObserver*	Removes *anObserver* as the observer of any notifications from any objects.
– (void)**removeObserver:**(id)*anObserver* **name:**(NSString *)*aName* **object:***anObject*	Removes *anObserver* as the observer of *aName* notifications from *anObject*.

Posting Notifications

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

Posts *aNotification* to the notification center. Raises NSInvalidArgumentException if the name associated with *aNotification* is **nil**.

– (void)**postNotificationName:**(NSString *)*aName* **object:**(id)*anObject*

Creates a notification object that associates *aName* and *anObject* and posts it to the notification center.

– (void)**postNotificationName:**(NSString *)*aName* **object:**(id)*anObject* **userInfo:**(NSDictionary *)*userInfo*

Creates a notification object that associates *aName* and *anObject* and posts it to the notification center. *userInfo* is a dictionary of arbitrary data that will be passed with the notification. *userInfo* may be **nil**.

NSNotificationQueue

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSNotificationQueue.h

Class Description

NSNotificationQueue objects (or simply, *notification queues*) act as buffers for notifications centers (instances of NSNotificationCenter). A notification queue maintains notifications (instances of NSNotification) generally in a FIFO order (First In First Out). When a notification rises to the "top" of the queue, the queue posts it to the notification center, which in turn dispatches the notification to all objects registered as observers.

NSNotificationQueue contributes two important features to OpenStep's notification mechanism: asynchronous posting and the coalescing of notifications. With NSNotificationCenter's **postNotification:** and its variants, you can post a notification immediately to a notification center. However, the invocation of the method is synchronous: Before the posting object can resume its thread of execution, it must wait until the notification center dispatches the notification to all observers and returns. With NSNotificationQueue's **enqueueNotification:postingStyle:** and **enqueueNotification:postingStyle:coalesceMask:forModes**:, however, you can post a notification asynchronously by putting it on the queue. These methods immediately return to the invoking object after putting the notification in the queue.

Posting to a notification queue can occur in one of three different styles. The posting style is an argument to both **enqueueNotification:...** methods:

- **NSPostWhenIdle**. The notification is posted when the run loop is idle.
- **NSPostASAP**. The notification is posted as soon as possible.
- **NSPostNow**. The notification is posted immediately to the notification center.

Note: See "Enqueuing with the Different Posting Styles," below, for details on and examples of enqueuing notifications with the three **postingStyle:** constants.

What is the difference between enqueuing notifications with **NSPostNow** and posting notifications (**postNotification:**)? Both post notifications immediately (but synchronously) to the notification center. The difference is that **enqueueNotification:...** (with **NSPostNow** as posting style) coalesces notifications in the queue before posting while **postNotification:** does not.

Coalescing is a process that removes notifications in the queue that are similar to the notification just enqueued (or posted, if posting style is **NSPostNow**. The notification queue scans the notifications in the queue for those with attributes matching the new notification and removes them, except for the notification that is topmost in the queue (closest to being posted). You indicate the criteria for similarity by specifying the NSNotificationCoalescing constants in the third argument of **enqueueNotification:postingStyle:coalesceMask:forModes**: (OR them in if multiple):

- **NSNotificationNoCoalescing**. Do not coalesce notifications in the queue.
- **NSNotificationCoalescingOnName**. Coalesce notifications with the same name.
- **NSNotificationCoalescingOnSender**. Coalesce notifications with the same sender.

Every task has a default notification queue, which is associated with the task's default notification center. You can create your own notification queues, and have multiple queues per center and task; but you can have only one notification center per task. NSNotificationQueue is a public, concrete class; instances of it are mutable.

Enqueuing with the Different Posting Styles

Any notification enqueued with the **NSPostASAP** posting style is posted to the notification center when the code executing in the current run loop callout completes. Callouts can be Application Kit event messages, file descriptor changes, timers, or another asynchronous notification. You'd typically use the **NSPostASAP** posting style for an expensive resource, like the Display PostScript server. When many clients draw on the window buffer during a callout, it's expensive to flush the buffer to the Display PostScript server after every draw operation. So in this case, each **draw...** method enqueues some notification such as "FlushTheServer" with coalescing on name and sender specified, and a posting style of **NSPostASAP**. As a result, only one of those notifications is dispatched at the end of the current callout, and the window buffer is flushed only once.

A notification enqueued with the **NSPostIdle** posting style is posted only when the run loop is in a wait state. In this state, there is nothing in the run loop's input channels, be it timers or other asynchronous notifications. A typical example of enqueuing with the **NSPostIdle** posting style occurs when the user types text, and the program displays the size of the text in bytes somewhere. It would be very expensive (and not very useful) to update the displayed size after each character the user types, especially if the user types fast. In this case, the program enqueues a notification after each character typed such as "ChangeTheDisplayedSize" with coalescing turned on and a posting style of **NSPostWhenIdle**. When the user stops typing, the single "ChangeTheDisplayedSize" notification in the queue (due to coalescing) is posted when the run loop is in a wait state and the display is updated.

A notification enqueued with **NSPostNow** is posted immediately to the notification center. You enqueue a notification with **NSPostNow** (or post one with NSNotificationCenter's **postNotification:**) when you do not require asynchronous calling behavior. For many programming situations, synchronous behavior is not only allowable but desirable; you want the notification center to return after dispatching so you can be sure that observing objects have received the notification. Of course, you should enqueue with **NSPostNow** rather than use **postNotification:** when there are similar notifications in the queue that you want to remove through coalescing.

Creating Notification Queues

+ (NSNotificationQueue *)**defaultQueue**

Returns the default NSNotificationQueue object for the current task. This object always uses the default notification-center object for the same task.

– (id)**init**

Initializes and returns an NSNotificationQueue object that uses the default notification-center object.

– (id)**initWithNotificationCenter:**(NSNotificationCenter *)*notificationCenter*

Initializes and returns an NSNotificationQueue object that uses the notification-center object specified in *notificationCenter.*

Inserting and Removing Notifications From a Queue

– (void)**dequeueNotificationsMatching:**(NSNotification *)*notification*
coalesceMask:(unsigned int)*coalesceMask*

Removes all notifications from the queue that match the *notification*'s attributes as specified by *coalesceMask*. The mask (set through NSNotificationCoalescing constants) can specify notification name, notification sender, or both name and sender.

– (void)**enqueueNotification:**(NSNotification *)*notification*
postingStyle:(NSPostingStyle)*postingStyle*

Puts a *notification* in the queue that the queue will post to the notification center at the time indicated by *postingStyle.* The notification queue posts in all runloop modes, and it coalesces only notifications in the queue that match both the name and sender of *notification*

– (void)**enqueueNotification:**(NSNotification *)*notification*
postingStyle:(NSPostingStyle)*postingStyle*
coalesceMask:(unsigned int)*coalesceMask*
forModes:(NSArray *)*modes*

Puts a *notification* in the queue that the queue will post to the notification center at the time indicated by *postingStyle,* but only if the runloop is in a mode identified by one of the string objects in the *modes* array. The notification queue coalesces related notifications in the queue as specified by *coalesceMask*. If *modes* is **nil**, all runloop modes are valid for posting.

NSNumber

Inherits From: NSValue : NSObject

Conforms To: NSCoding, NSCopying (NSValue)
NSObject (NSObject)

Declared In: Foundation/NSValue.h

Class Description

NSNumber objects provide an object-oriented wrapper for the standard C-language number data types (**int**, **double**, etc.). The Foundation Kit's collection classes can store only objects, so this class provides a way to prepare numbers of various types for use with the collection classes.

NSNumber, which inherits from NSValue, provides methods for creating number objects that contain data of a specified type. It also provides methods for extracting data from a number object and casting the data to be of a particular type. For determining whether two number objects are equal, NSNumber provides the **compare:** method.

Allocating and Initializing

+ (NSNumber *)**numberWithBool:**(BOOL)*value* — Creates and returns a number object representing *value* of the type **BOOL**.

+ (NSNumber *)**numberWithChar:**(char)*value* — Creates and returns a number object representing *value* of the type **char**.

+ (NSNumber *)**numberWithDouble:**(double)*value* — Creates and returns a number object representing *value* of the type **double**.

+ (NSNumber *)**numberWithFloat:**(float)*value* — Creates and returns a number object representing *value* of the type **float**.

+ (NSNumber *)**numberWithInt:**(int)*value* — Creates and returns a number object representing *value* of the type **int**.

+ (NSNumber *)**numberWithLong:**(long)*value* — Creates and returns a number object representing *value* of the type **long**.

+ (NSNumber *)**numberWithLongLong:**(long long)*value* — Creates and returns a number object representing *value* of the type **long long**.

+ (NSNumber *)**numberWithShort:**(short)*value* — Creates and returns a number object representing *value* of the type **short**.

+ (NSNumber *)**numberWithUnsignedChar:**(unsigned char)*value*

Creates and returns a number object representing *value* of the type **unsigned char**.

+ (NSNumber *)**numberWithUnsignedInt:**(unsigned int)*value*

Creates and returns a number object representing *value* of the type **unsigned int**.

+ (NSNumber *)**numberWithUnsignedLong:**(unsigned long)*value*

Creates and returns a number object representing *value* of the type **unsigned long**.

+ (NSNumber *)**numberWithUnsignedLongLong:**(unsigned long long)*value*

Creates and returns a number object representing *value* of the type **unsigned long long**.

+ (NSNumber *)**numberWithUnsignedShort:**(unsigned short)*value*

Creates and returns a number object representing *value* of the type **unsigned short**.

Accessing Data

– (BOOL)**boolValue**

Returns the receiver's value as a boolean value.

– (char)**charValue**

Returns the receiver's value as a character value.

– (double)**doubleValue**

Returns the receiver's value as a double precision floating point value.

– (float)**floatValue**

Returns the receiver's value as a single precision floating point value.

– (int)**intValue**

Returns the receiver's value as a integer value.

– (long long)**longLongValue**

Returns the receiver's value as a long long double precision floating point value.

– (long)**longValue**

Returns the receiver's value as a long double precision floating point value.

– (short)**shortValue**

Returns the receiver's value as a short integer value.

– (NSString *)**stringValue**

Returns the receiver's value as a string contained in an NSString object.

– (unsigned char)**unsignedCharValue**

Returns the receiver's value as an unsigned character value.

– (unsigned int)**unsignedIntValue**

Returns the receiver's value as an unsigned integer value.

– (unsigned long long)**unsignedLongLongValue**

Returns the receiver's value as an unsigned long long double precision floating point value.

– (unsigned long)**unsignedLongValue**

Returns the receiver's value as an unsigned long double precision floating point value.

– (unsigned short)**unsignedShortValue**

Returns the receiver's value as an unsigned short integer value.

Comparing Data

– (NSComparisonResult)**compare:**(NSNumber *)*otherNumber*

Compares the receiver to *otherNumber*, using ANSI C rules for type coersion, and returns an NSComparisonResult.

NSObject

Inherits From: none *(NSObject is the root class)*

Conforms To: NSObject

Declared In: Foundation/NSObject.h
Foundation/NSRunLoop.h

Class Description

NSObject is the root class of all ordinary Objective C inheritance hierarchies; it has no superclass. Its interface derives from two sources: the methods it declares directly and those declared in the NSObject protocol. Its interface is divided in this way so that objects inheriting from other root classes (notably NSProxy) can stand in for ordinary objects without having to inherit from NSObject. The following discussion makes no distinction between the methods declared in this class and those declared in the NSObject protocol.

From NSObject, other classes inherit a basic interface to the run-time system for the Objective C language. It's through NSObject that instances of all classes obtain their ability to behave as objects. Among other things, the NSObject class provides inheriting classes with a framework for creating, initializing, deallocating, comparing, and archiving objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy, and for forwarding messages to other objects. For example, to ask an object what class it belongs to, you'd send it a **class** message. To find out whether it implements a particular method, you'd send it a **respondsToSelector:** message

The NSObject class is an abstract class; programs use instances of classes that inherit from NSObject, but never of NSObject itself.

Initializing an Object to Its Class

Every object is connected to the run-time system through its **isa** instance variable, inherited from the NSObject class. **isa** identifies the object's class; it points to a structure that's compiled from the class definition. Through **isa**, an object can find whatever information it needs at run time—such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages.

Because all ordinary objects inherit directly or indirectly from the NSObject class, they all have this variable. The defining characteristic of an "object" is that its first instance variable is an **isa** pointer to a class structure.

The installation of the class structure—the initialization of **isa**—is one of the responsibilities of the **alloc** and **allocWithZone:** methods, the same methods that create (allocate memory for) new instances of a class. In other words, class initialization is part of the process of creating an object; it's not left to the methods, such as **init**, that initialize individual objects with their particular characteristics.

Instance and Class Methods

Every object requires an interface to the run-time system, whether it's an instance object or a class object. For example, it should be possible to ask either an instance or a class whether it can respond to a particular message. So that this won't mean implementing every NSObject method twice, once as an instance method and again as a class method, the run-time system treats methods defined in the root class in a special way:

> *Instance methods defined in the root class can be performed both by instances and by class objects.*

A class object has access to class methods—those defined in the class and those inherited from the classes above it in the inheritance hierarchy—but generally not to instance methods. However, the run-time system gives all class objects access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it doesn't have a class method with the same name.

For example, a class object could be sent messages to perform NSObject's **respondsToSelector:** and **perform:withObject:** instance methods:

```
SEL method = @selector(riskAll:);

if ( [MyClass respondsToSelector:method] )
    [MyClass perform:method withObject:self];
```

When a class object receives a message, the run-time system looks first at the receiver's set of class methods. If it fails to find a class method that can respond to the message, it looks at the set of instance methods defined in the root class. If the root class has an instance method that can respond (as NSObject does for **respondsToSelector:** and **perform:withObject:**), the run-time system uses that implementation and the message succeeds.

Note that the only instance methods available to a class object are those defined in the root class. If MyClass in the example above had reimplemented either **respondsToSelector:** or **perform:withObject:**, those new versions of the methods would be available only to instances. The class object for MyClass could perform only the versions defined in the NSObject class. (Of course, if MyClass had implemented **respondsToSelector:** or **perform:withObject:** as class methods rather than instance methods, the class would perform those new versions.)

Initializing the Class

+ (void)**initialize**	Initializes the class before it's used (before it receives its first message).

Creating and Destroying Instances

+ (id)**alloc**	Returns a new, uninitialized instance of the receiving class.
+ (id)**allocWithZone:**(NSZone *)*zone*	Returns a new, uninitialized instance of the receiving class in *zone*.

+ (id)**new**

Allocates a new instance of the receiving class, sends it an **init** message, and returns the initialized object returned by **init**. This method is simply a convenient cover for the **alloc** and **init** methods.

– (id)**copy**

Invokes **copyWithZone:**. This method is implemented in NSObject as a convenience to subclasses. A subclass need override only **copyWithZone:** for both **copy** and **copyWithZone:** to operate correctly.

– (void)**dealloc**

Deallocates the memory occupied by the receiver.

– (id)**init**

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated.

– (id)**mutableCopy**

Invokes **mutableCopyWithZone:**. This method is implemented in NSObject as a convenience to subclasses. A subclass need override only **mutableCopyWithZone:** for both **mutableCopy** and **mutableCopyWithZone:** to operate correctly.

Identifying Classes

+ (Class)**class**

Returns **self**. Since this is a class method, it returns the class object.

+ (Class)**superclass**

Returns the class object for the receiver's superclass.

Testing Class Functionality

+ (BOOL)**instancesRespondToSelector:**(SEL)*aSelector*

Returns YES if instances of the class are capable of responding to *aSelector* messages, and NO if they're not.

Testing Protocol Conformance

+ (BOOL)**conformsToProtocol:**(Protocol *)*aProtocol*

Returns YES if the receiving class conforms to *aProtocol*, and NO if it doesn't.

Obtaining Method Information

+ (IMP)**instanceMethodForSelector:**(SEL)*aSelector*

Locates and returns the address of the implementation of the *aSelector* instance method.

– (IMP)**methodForSelector:**(SEL)*aSelector*

Locates and returns the address of the receiver's implementation of the *aSelector* method, so that it can be called as a function.

– (NSMethodSignature *)**methodSignatureForSelector:**(SEL)*aSelector*

Returns an object that contains a description of the *aSelector* method, or **nil** if the *aSelector* method can't be found.

Describing Objects

+ (NSString *)**description**

Subclasses override this method to return a human-readable string representation of the contents of the receiver. NSObject's implementation simply prints the name of the receiver's class.

Posing

+ (void)**poseAsClass:**(Class)*aClass*

Causes the receiving class to "pose as" its superclass.

Error Handling

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

Handles *aSelector* messages that the receiver doesn't recognize.

Sending Deferred Messages

+ (void)**cancelPreviousPerformRequestsWithTarget:**(id)*aTarget*
selector:(SEL)*aSelector*
object:(id)*anObject*

Cancels previous perform requests having the same target and argument (as determined by **isEqual:**), and the same selector. This method removes timers only in the current run loop, not all run loops.

– (void)**performSelector:**(SEL)*aSelector*
object:(id)*anObject*
afterDelay:(NSTimeInterval)*delay*

Sends an *aSelector* message to *anObject* after *delay*. **self** and *anObject* are retained until after the action is executed.

Forwarding Messages

– (void)**forwardInvocation:**(NSInvocation *)*anInvocation*
Implemented by subclasses to forward messages to other objects.

Archiving

– (id)**awakeAfterUsingCoder:**(NSCoder *)*aDecoder* Implemented by subclasses to reinitialize the receiver. The NSObject implementation of this method simply returns **self**.

– (Class)**classForArchiver** Identifies the class to be used during archiving. NSObject's implementation returns the object returned by **classForCoder:**.

– (Class)**classForCoder** Identifies the class to be used during serialization. An NSObject returns its own class by default.

– (id)**replacementObjectForArchiver:**(NSArchiver *)*anArchiver*
Allows an object to substitute another object for itself during archiving. NSObject's implementation returns the object returned by **replacementObjectForCoder:**.

– (id)**replacementObjectForCoder:**(NSCoder *)*anEncoder*
Allows an object to substitute another object for itself during serialization. NSObject's implementation returns **self**.

\+ (void)**setVersion:**(int)*version* Sets the class version number to *version*.

\+ (int)**version** Returns the version of the class definition.

NSProcessInfo

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSProcessinfo.h

Class Description

The NSProcessInfo class provides methods to access process-wide information. An NSProcessInfo object can return such information as the arguments, environment, host name, or process name. The **processInfo** class method returns an NSProcessInfo object. For example, the following code creates an NSProcessInfo object, which then provides the name of the current process:

```
[[NSProcessInfo processInfo] processName];
```

Getting an NSProcessInfo Object

+ (NSProcessInfo *)**processInfo**
Returns the NSProcessInfo object for the process. It is already initialized. An NSProcessInfo object is created the first time this method is invoked, and that same object is returned on each subsequent invocation.

Returning Process Information

– (NSArray *)**arguments**
Returns the arguments as an array of NSStrings from the command line.

– (NSDictionary *)**environment**
Returns a dictionary of variables defined for the environment from which the process was launched.

– (NSString *)**hostName**
Returns the name of the host system.

– (NSString *)**processName**
Returns the name of the process under which this program's user defaults domain is created, and is the name used in error messages. It does not uniquely identify the process.

– (NSString *)**globallyUniqueString**
Returns a globally unique string to identify the process. This method uses the host name, process ID, and a timestamp to ensure that the string returned will be globally unique.

Specifying a Process Name

– (void)**setProcessName:**(NSString *)*newName*

Sets the name of the process to *newName*. Warning: Aspects of the environment like user defaults might depend on the process name, so be very careful if you change this. Setting the process name this way is not thread-safe.

NSProxy

Inherits From: none (*NXProxy is a root class*)

Conforms To: NSObject

Declared In: Foundation/NSProxy

Class Description

The NSProxy class declares the programmatic interface to *proxies*—objects that stand in for real objects (usually descendants of the NSObject class), where the real objects may exist within the same or another process, perhaps even in a system of a different architecture across a network. To the application, the proxy behaves like the real object, though the real object may not be directly accessible, and in general, instance variables of remote objects are not accessible.

NSProxy class defines few methods, because proxies respond to few messages directly. Instead, when a proxy receives a message it doesn't respond to, it encodes the message, including the arguments, in an invocation, and invokes **forwardInvocation:**. Specialized subclasses then direct further processing, such as forwarding the message to a real object in the same or another process.

Methods defined in this class are methods that the NSProxy class responds to directly. Unless otherwise noted, none of these methods are forwarded to the proxy's correspondent.

Your application in general doesn't instantiate NSProxy objects—they're created as instances of specialized subclasses. Proxies are reference-counted so that only a single NSProxy per connection is instantiated for any real object.

Creating and Destroying Instances

+ (id)**alloc**	Returns a new, uninitialized instance of the receiving class.
+ (id)**allocWithZone:**(NSZone *)*zone*	Returns a new, uninitialized instance of the receiving class in *zone*.
– (void)**dealloc**	Deallocates the memory occupied by the receiver.

Identifying Classes

+ (Class)**class**	Returns **self**. Since this is a class method, it returns the class object.

Obtaining Method Information

– (NSMethodSignature *)**methodSignatureForSelector:**(SEL)*aSelector*

Implemented by subclasses to return an object that contains a description of the *aSelector* method, or **nil** if the *aSelector* method can't be found. The NSProxy implementation of this method raises an NSInvalidArgumentException exception.

Describing Objects

– (NSString *)**description**

Prints the name of receiver's class and the hexadecimal value of the its **id**.

Forwarding Messages

– (void)**forwardInvocation:**(NSInvocation *)*invocation*

Implemented by subclasses to forward messages to other objects. The NSProxy implementation of this method raises an NSInvalidArgumentException exception.

NSRecursiveLock

Inherits From: NSObject

Conforms To: NSLocking
NSObject (NSObject)

Declared In: Foundation/NSLock.h

Class Description

NSRecursiveLock is used for locks that need to be reacquired by the same thread.

An NSRecursiveLock locks a critical section of code such that a single thread can reaquire the lock multiple times without deadlocking, while preventing access by other threads. (Note that this implies that a recursive lock will not protect a critical section from a signal handler interrupting the thread holding the lock.) Here is an example where a recursive lock functions properly but other lock types would deadlock:

```
// create the lock only once!
NSRecursiveLock  *theLock = [NSRecursiveLock new];
/* ...other code... */
[theLock lock];

/* ... possibly a long time of fussing with global data... */
[theLock lock]; /* possibly invoked in a subroutine */
[theLock unlock];

[theLock unlock];
```

The NSConditionLock, NSLock, and NSRecursiveLock classes all implement the NSLocking protocol with various features and performance characteristics; see the other class descriptions for more information.

Acquiring a Lock

– (BOOL)**tryLock** Attempts to acquire a lock. Returns YES if successful and NO otherwise. This method can be called repeatedly to produce nested locks.

NSRunLoop

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSRunLoop.h

Class Description

The NSRunLoop class declares the programmatic interface to objects that manage input sources. An NSRunLoop object processes input for sources such as mouse and keyboard events from the window system, NSTimers, POSIX file descriptors, and NSConnections, based on a *mode* argument. A given NSRunLoop object processes input for input sources associated with a particular mode.

In general, your application won't need to either create or explicitly manage NSRunLoop objects. Each thread has an NSRunLoop object automatically created for it. The NSApplication object creates a default thread and therefore creates a default run loop.

Applications wanting to perform their own explicit run loop management should send the **currentRunLoop** message to the NSRunLoop class object to obtain the NSRunLoop object for the current thread, then invoke one of the methods described below in "Running a Run Loop" to obtain input.

Currently defined modes are:

NSDefaultRunLoopMode	Use this mode to deal with input sources other than NSConnections. Defined in the **Foundation/NSRunLoop.h** header file.
NSConnectionReplyMode	Use this mode to indicate NSConnections waiting for replies. Defined in the **Foundation/NSConnection.h** header file.

Accessing the Current Run Loop

+ (NSRunLoop *)**currentRunLoop**	Returns the NSRunLoop for the current thread.
– (NSString *)**currentMode**	Returns the current run loop mode.
– (NSDate *)**limitDateForMode:**(NSString *)*mode*	Polls timers and platform-specific input managers for their limit date (if any). Timers will fire if appropriate. Returns **nil** if there are no input sources for this mode.

Adding Timers

– (void)**addTimer:**(NSTimer *)*aTimer*
forMode:(NSString *)*mode*

Registers the timer *aTimer* with input filter *mode*. The run loop causes the timer to fire at its scheduled fire date. Note that timers are removed from modes if they supply **nil** as their fire date.

Running a Run Loop

– (void)**acceptInputForMode:**(NSString *)*mode*
beforeDate:(NSDate *)*limitDate*

Runs the run loop, accepting input from the input sources for the mode specified by *mode* until the time specified by *limitDate*.

– (void)**run**

Runs the run loop in the default mode until there is nothing to do.

– (BOOL)**runMode:**(NSString *)*mode*
beforeDate:(NSDate *)*limitDate*

Runs the run loop, accepting input from filter *mode* until *limitDate* or until the earliest limit date for input sources in this mode. Returns NO without starting the run loop if there are no limit dates set for input sources (that is, there's nothing to do).

– (void)**runUntilDate:**(NSDate *)*limitDate*

Runs the run loop until *limitDate* or until there are no limit dates set for input sources (that is, there's nothing to do).

NSScanner

Inherits From: NSObject

Conforms To: NSCopying
NSObject (NSObject)

Declared In: Foundation/NSScanner.h

Class Description

The NSScanner class declares the programmatic interface to an object that is capable of scanning NSString objects (strings of characters in the Unicode character encoding), con verting the scanned strings to various numeric representations, or scanning characters from a character set.

Generally, you instantiate a scanner object by sending one of **scannerWithString**: or **localizedScannerWithString**: methods to the NSScanner class object. Either method returns a scanner object initialized with the string you pass in.

NSScanner provides methods of configuring the behavior of the scan. **setCaseSensitive:** specifies whether the scanner will treat upper case and lower case letters as distinct. **setCharactersToBeSkipped:** determines the set of characters that will be skipped while scanning. The preset set of characters to skip are whitespace and newline characters. **setLocale:** specifies the locale to be used while scanning strings**. setScanLocation:** sets the index in the string object at that scanning will commence. Using this method, you can repeatedly scan portions of a string.

Scanning is performed using any of the **scan**… methods listed under “Scanning a String”.

Note that floating point numbers are assumed to be IEEE compliant.

Creating an NSScanner

+ (id)**localizedScannerWithString:**(NSString *)*aString*
Creates and returns a scanner that scans *aString*. Invokes **initWithString:** and sets the locale to the user’s default locale.

+ (id)**scannerWithString:**(NSString *)*aString* — Creates and returns a scanner that scans *aString*.

– (id)**initWithString:**(NSString *)*aString* — Initializes the receiver, a newly allocated scanner, to scan *aString*. Returns **self**.

Getting an NSScanner’s String

– (NSString *)**string** — Returns the string object that the scanner was created with.

Configuring an NSScanner

– (BOOL)**caseSensitive**
Returns YES if the scanner distinguishes case, and NO otherwise. Scanners are by default *not* case sensitive.

– (NSCharacterSet *)**charactersToBeSkipped**
Returns a character set object containing those characters that the scanner ignores when looking for an element. The default set is the whitespace and newline character set.

– (NSDictionary *)**locale**
Returns a dictionary object containing locale information. Returns **nil** if the locale dictionary has not been set.

– (unsigned)**scanLocation**
Returns the character index at which the scanner will begin its next scanning operation.

– (void)**setCaseSensitive:**(BOOL)*flag*
If *flag* is YES, the scanner considers case when scanning characters. If *flag* is NO, it ignores case distinctions. NSScanners are by default *not* case sensitive.

– (void)**setCharactersToBeSkipped:**(NSCharacterSet *)*aSet*
Sets the scanner to ignore characters from *aSet* when scanning its string.

– (void)**setLocale:**(NSDictionary *)*localeDictionary*
Sets the receiver's dictionary object containing locale information.

– (void)**setScanLocation:**(unsigned int)*anIndex*
Sets the location at which the next scan will begin to *anIndex*.

Scanning a String

In the **scan**…methods listed here, the *value* arguments (which are values returned by reference) are optional. Pass an argument value of **nil** if you do not wish a return value.

– (BOOL)**scanCharactersFromSet:**(NSCharacterSet *)*aSet*
intoString:(NSString **)*value*
Scans the string as long as characters from *aSet* are encountered, accumulating characters into an optional string that's returned by reference in *value*. If any characters are scanned, returns YES; otherwise returns NO.

– (BOOL)**scanDouble:**(double *)*value*
Scans a **double** into *value* if possible. Returns YES if a valid floating-point expression was scanned; NO otherwise. HUGE_VAL or –HUGE_VAL is put in *value* on overflow; 0.0 on underflow. Returns YES in overflow and underflow cases

– (BOOL)**scanFloat:**(float *)*value*

Scans a **float** into *value* if possible. Returns YES if a valid floating-point expression was scanned; NO otherwise. HUGE_VAL or –HUGE_VAL is put in *value* on overflow; 0.0 on underflow. Returns YES in overflow and underflow cases.

– (BOOL)**scanInt:**(int *)*value*

Scans an **int** into *value* if possible. Returns YES if a valid integer expression was scanned; NO otherwise. INT_MAX or INT_MIN is put in *value* on overflow. Returns YES in overflow cases.

– (BOOL)**scanLongLong:**(long long *)*value*

Scans a **long long int** into *value* if possible. Returns YES if a valid integer expression was scanned; NO otherwise. LONG_LONG_MAX or LONG_LONG_MIN is put in *value* on overflow. Returns YES in overflow cases.

– (BOOL)**scanString:**(NSString *)*aString* **intoString:**(NSString **)*value*

Scans for *aString*, and if a match is found returns by reference in the optional *value* argument a string object equal to it. If *aString* matches the characters at the scan location, returns YES; otherwise returns NO.

– (BOOL)**scanUpToCharactersFromSet:**(NSCharacterSet *)*aSet* **intoString:**(NSString **)*value*

Scans the string until a character from *aSet* is encountered, accumulating characters encountered into a string that's returned by reference in the optional *value* argument. If any characters are scanned, returns YES; otherwise returns NO.

– (BOOL)**scanUpToString:**(NSString *)*aString* **intoString:**(NSString **)*value*

Scans the string until *aString* is encountered, accumulating characters encountered into a string that's returned by reference in the optional *value* argument. If any characters are scanned, returns YES; otherwise returns NO.

– (BOOL)**isAtEnd**

Returns YES if the scanner has exhausted all characters in its string; NO if there are characters left to scan.

NSSerializer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSSerialization.h

Class Description

The NSSerializer class provides a mechanism for creating an abstract representation of a property list. (In OpenStep, property lists are defined to be—and to contain—objects of these classes: NSDictionary, NSArray, NSString, NSData). The NSSerializer class stores this representation in an NSData object in an architecture-independent format, so that property lists can be used with distributed applications. NSSerializer's companion class NSDeserializer declares methods that take the abstract representation and recreate the property list in memory.

In contrast to archiving (see the NSArchiver class specification), the serialization process preserves only structural information, not class information. Thus, if a property list is serialized and then deserialized, the objects in the resulting property list might not be of the same class as the objects in the original property list. However, the structure and interrelationships of the data in the resulting property list are identical to that in the original, with one possible exception.

The exception is that when an object graph is serialized, the mutability of the containers objects (NSDictionary and NSArray objects) is preserved only down to the highest node in the graph that has an immutable container. Thus, if an NSArray contains an NSMutableDictionary, the serialized version of this object graph would not preserve the mutability of the dictionary or any of the mutable objects it contained. Since serialization doesn't preserve class information or—in some cases—mutability, coding (as implemented by NSCoder and NSArchiver) is the preferred way to make object graphs persistent.

The NSSerializer class object provides the interface to the serialization process; you don't create instances of NSSerializer. You might subclass NSSerializer to modify the representation it creates, for example, to encrypt the data or add authentication information.

Other types of data besides property lists can be serialized using methods declared by the NSData and NSMutableData classes (see **serializeDataAt:ofObjCType:context:** and **deserializeDataAt:ofObjCType:atCursor:context:**), allowing these types to be represented in an architecture-independent format. Furthermore, the NSObjCTypeSerializationCallBack protocol allows you to serialize and deserialize objects that aren't property lists.

Serialization of Property Lists

+ (NSData *)**serializePropertyList:**(id)*aPropertyList*

Creates a data object, serializes *aPropertyList* into it, and returns the data object. *aPropertyList* must be a kind of NSData, NSString, NSArray, or NSDictionary.

+ (void)**serializePropertyList:**(id)*aPropertyList* **intoData:**(NSMutableData *)*mdata*

Serializes the property list *aPropertyList* in the mutable data object *mdata*. *aPropertyList* must be a kind of NSData, NSString, NSArray, or NSDictionary.

NSSet

Inherits From: NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying
NSObject (NSObject)

Declared In: Foundation/NSSet.h

Class Description

The NSSet class declares the programmatic interface to an object that manages an immutable set of objects. NSSet provides support for the mathematical concept of a *set*. A set, both in its mathematical sense and in the OpenStep implementation of NSSet, is an *unordered* collection of distinct elements. OpenStep provides the NSMutableSet class for sets whose contents may be altered, and also provides the NSCountedSet class for sets that can contain multiple instances of the same element.

Use set objects as an alternative to array objects when the order of elements is not important, but performance in testing whether an object is contained in the set *is* a consideration—while arrays are ordered, testing for membership is slower than with sets. For example, the NSSet method **containsObject:** operates in O(1) time when applied to a set, while **containsObject:** operates in O(N) time when applied to an array.

Objects in a set must respond to **hash** and **isEqual:** methods. See the NSObject protocol for details on **hash** and **isEqual:**.

Generally, you instantiate an NSSet object by sending one of the **set…** methods to the NSSet class object. These methods return an NSSet object containing the elements (if any) you pass in as arguments. The **set** method is a "convenience" method to create an empty set. Newly created instances of NSSet created by invoking the **set** method can be populated with objects using any of the **init…** methods. **initWithObjects::** is the designated initializer for the NSSet class. Objects added to the set are not copied; rather, each object receives a **retain** message before it's added to the set.

NSSet provides methods for querying the elements of the set. **allObjects** returns an array containing all objects in the set. **anyObject** returns some object in the set. **count** returns the number of objects currently in the set. **member:** returns the object in the set that is equal to a specified object. Additionally, the **intersectsSet:** tests for set intersection, **isEqualToSet:** tests for set equality, and **isSubsetOfSet:** tests for one set being a subset of the specified set object.

The **objectEnumerator** method provides for traversing elements of the set one by one.

NSSet's **makeObjectsPerform:** and **makeObjectsPerform:withObject:** methods provides for sending messages to individual objects in the set.

Exceptions

NSSet implements the **encodeWithCoder:** method, which raises NSInternalInconsistencyException if the number of objects enumerated for encoding turns out to be unequal to the number of objects in the set.

Allocating and Initializing a Set

+ (id)**allocWithZone:**(NSZone *)*zone*	Creates and returns an uninitialized set object in *zone*.
+ (id)**set**	Creates and returns an empty set object.
+ (id)**setWithArray:**(NSArray *)*array*	Creates and returns a set object containing the objects in *array.*
+ (id)**setWithObject:**(id)*anObject*	Creates and returns a set object containing the single element *anObject.*
+ (id)**setWithObjects:**(id)*firstObj*,...	Creates and returns a set object containing the objects in the argument list. The object list is comma-separated and ends with **nil**.
– (id)**initWithArray:**(NSArray *)*array*	Initializes a newly allocated set object by placing in it the objects contained in *array.*
– (id)**initWithObjects:**(id)*firstObj*,...	Initializes a newly allocated set object by placing in it the objects in the argument list. The object list is comma-separated and ends with **nil**.
– (id)**initWithObjects:**(id *)*objects* **count:**(unsigned int)*count*	Initializes a newly allocated set object by placing in it *count* objects from the *objects* array.
– (id)**initWithSet:**(NSSet *)*anotherSet*	Initializes a newly allocated set object by placing in it the objects contained in *anotherSet.*
– (id)**initWithSet:**(NSSet *)*set* **copyItems:**(BOOL)*flag*	Initializes a newly allocated set object by placing in it the objects contained in *anotherSet* (or immutable copies of them, if *flag* is YES).

Querying the Set

– (NSArray *)**allObjects**	Returns an array containing all the objects in the set.
– (id)**anyObject**	Returns some object in the set, or **nil** if the set is empty.
– (BOOL)**containsObject:**(id)*anObject*	Returns YES if *anObject* is present in the set.
– (unsigned int)**count**	Returns the number of objects currently in the set.
– (id)**member:**(id)*anObject*	Return the object in the set that is equal to *anObject*, or **nil** if none is equal.
– (NSEnumerator *)**objectEnumerator**	Returns an enumerator object that lets you access each object in the set.

Sending Messages to Elements of the Set

– (void)**makeObjectsPerform:**(SEL)*aSelector*	Sends an *aSelector* message to each object in the set.
– (void)**makeObjectsPerform:**(SEL)*aSelector* **withObject:**(id)*anObject*	Sends an *aSelector* message to each object in the set, with *anObject* as an argument.

Comparing Sets

– (BOOL)**intersectsSet:**(NSSet *)*otherSet*	Returns YES if there's any object in the receiving set that's equal to an object in *otherSet*.
– (BOOL)**isEqualToSet:**(NSSet *)*otherSet*	Returns YES if every object in the receiving set is equal to an object in *otherSet*, and the two sets contain the same number of objects.
– (BOOL)**isSubsetOfSet:**(NSSet *)*otherSet*	Returns YES if every object in the receiving set is equal to an object in *otherSet*, and the receiving set contains no more objects than *otherSet* does.

Creating a String Description of the Set

– (NSString *)**description**	Returns a string object that describes the contents of the receiver.
– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary*	Returns a string representation of the NSSet object, including the keys and values that represent the locale data from *localeDictionary.*

NSString

Inherits From: NSObject

Conforms To: NSCoding, NSCopying, NSMutableCopying
NSObject (NSObject)

Declared In: Foundation/NSString.h
Foundation/NSPathUtilities.h
Foundation/NSUtilities.h

Class Description

NSString declares the programmatic interface for objects that create and manage immutable character strings in a *representation-independent* format.

NSString (and NSMutableString) are abstract classes for string manipulation. NSString provides methods for read-only access, while NSMutableString allows for changing the contents of the string. NSString and NSMutableString provide factory methods that return autoreleased instances of unspecified subclasses of strings.

While the actual representation of character strings stored in NSString and NSMutableString is independent of any particular implementation, you can in general think of the contents of NSString and NSMutableString objects as being, canonically, **Unicode** characters (defined by the **unichar** data type). Methods that use the terms “character”, “range”, and “length”, refer to strings of **unichar**s and ranges and lengths of **unichar** strings—this is important, because conversion between **unichar**s and other character encodings is not necessarily one-to-one. For instance, an ISO Latin1 encoded string of a given length might contain fewer or more characters when encoded as **unichar**s. Another important point is that **unichar**s don't necessarily correspond one-to-one with what is normally thought of as “letters” in a string; if you need to go through a string in terms of ”letters”, use **rangeOfComposedCharacterSequenceAtIndex:**.

Methods that take “CString” arguments deal with the default eight-bit encoding of the environment, which could be, for instance, EUC or ISOLatin1. You can also explicitly convert to and from any encoding by using methods such as **initWithData:usingEncoding:** and **dataUsingEncoding:**.

Constant NSStrings can be created with the @"..." option—such strings should contain only ASCII characters, and nothing more.

Strings are provided with generic coding behavior when used for storage or distribution. This behavior is to copy the contents and provide a generic NSString implementation, losing class but preserving mutability.

In general, you instantiate NSString objects sending one of the **stringWith**… methods or the **localizedStringWithFormat:** method to the NSString class object. For NSString objects that were allocated “manually”, use any of the **initWith**… methods to initialize the contents of the string object.

The primitive methods to NSString are **length** and **characterAtIndex:**.

UNIX-style file system path names can be manipulated using the collection of **stringBy**… methods described under “Manipulating File System Paths” below.

Creating Temporary Strings

+ (NSString *)**localizedStringWithFormat:**(NSString *)*format*,...

Returns a string created by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string. The user’s default locale is used for format information.

+ (NSString *)**stringWithCString:**(const char *)*byteString*

Returns a string containing the characters in *byteString*, which must be null-terminated. *byteString* should contain characters in the default C string encoding.

+ (NSString *)**stringWithCString:**(const char *)*byteString* **length:**(unsigned int)*length*

Returns a string containing characters from *byteString*. *byteString* should contain characters in the default C string encoding. *length* bytes are copied into the string, regardless of whether a null byte exists in *byteString*. Raises NSInvalidArgumentException if *byteString* is NULL

+ (NSString *)**stringWithCharacters:**(const unichar *)*cha*rs **length:**(unsigned int)*length*

Returns a string containing *chars*. *length* characters are copied into the string, regardless of whether a null character exists in *chars*.

+ (NSString *)**stringWithContentsOfFile:**(NSString *)*path*

Returns a string containing the contents of the file specified by *path*. This method attempts to determine the encoding for the file. The string is assumed to be in Unicode encoding, but if the encoding is determined not to be Unicode, the default C string encoding is used instead.

+ (NSString *)**stringWithFormat:**(NSString *)*format*,...

Returns a string created by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string.

Initializing Newly Allocated Strings

– (id)**init**

Initializes the receiver, a newly allocated NSString, to contain no characters. This is the only initialization method that a subclass of NSString should invoke.

– (id)**initWithCString:**(const char *)*byteString*

Initializes the receiver, a newly allocated NSString, by converting the one-byte characters in *byteString* into Unicode characters. *byteString* must be a null-terminated C string in the default C string encoding.

– (id)**initWithCString:**(const char *)*byteString* **length:**(unsigned int)*length*

Initializes the receiver, a newly allocated NSString, by converting *length* one-byte characters in *byteString* into Unicode characters. This method doesn't stop at a null byte.

– (id)**initWithCStringNoCopy:**(char *)*byteString* **length:**(unsigned int)*length* **freeWhenDone:**(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, by converting *length* one-byte characters in *byteString* into Unicode characters. This method doesn't stop at a null byte. The receiver becomes the owner of *byteString;* if *flag* is YES it will free the memory when it no longer needs it, but if *flag* is NO it won't.

– (id)**initWithCharacters:**(const unichar *)*chars* **length:**(unsigned int)*length*

Initializes the receiver, a newly allocated NSString, by copying *length* characters from *chars*. This method doesn't stop at a null character.

– (id)**initWithCharactersNoCopy:**(unichar *)*chars* **length:**(unsigned int)*length* **freeWhenDone:**(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, to contain *length* characters from *chars*. This method doesn't stop at a null character. The receiver becomes the owner of *chars;* if *flag* is YES the receiver will free the memory when it no longer needs them, but if *flag* is NO it won't. Note that the NO case could be dangerous if used with memory that could be freed. The NO flag should be used only when the provided backing store is permanent.

– (id)**initWithContentsOfFile:**(NSString *)*path*

Initializes the receiver, a newly allocated NSString, by reading characters from the file whose name is given by *path*. This method attempts to determine the encoding for the file. The string is assumed to be in Unicode encoding, but if the encoding is determined not to be Unicode, the default C string encoding is used instead. Also see **writeToFile:atomically:** in “Storing the String”.

– (id)**initWithData:**(NSData *)*data* **encoding:**(NSStringEncoding)*encoding*

Initializes the receiver, a newly allocated NSString, by converting the bytes in *data* into Unicode characters. *data* must be an NSData object containing bytes in *encoding* and in the default “plain text” format for that encoding.

– (id)**initWithFormat:**(NSString *)*format,...*

Initializes the receiver, a newly allocated NSString, by constructing a string from *format* and following string objects in the manner of **printf()**.

– (id)**initWithFormat:**(NSString *)*format* **arguments:**(va_list)*argList*

Initializes the receiver, a newly allocated NSString, by constructing a string from *format* and *argList* in the manner of **vprintf()**.

– (id)**initWithFormat:**(NSString *)*format* **locale:**(NSDictionary *)*dictionary,...*

Initializes the receiver, a newly allocated NSString, by constructing a string from *format* and the formatting information in the dictionary in the manner of **printf()**.

– (id)**initWithFormat:**(NSString *)*format* **locale:**(NSDictionary *)*dictionary* **arguments:**(va_list)*argList*

Initializes the receiver, a newly allocated NSString, by constructing a string from *format* and format information in *dictionary* and *argList* in the manner of **vprintf()**.

– (id)**initWithString:**(NSString *)*string*

Initializes the receiver, a newly allocated NSString, by copying the characters from *string*.

Getting a String’s Length

– (unsigned int)**length**

Returns the number of characters in the receiver. This number includes the individual characters of composed character sequences.

Accessing Characters

– (unichar)**characterAtIndex:**(unsigned int)*index*

Returns the character at the array position given by *index*. This method raises an **NSStringBoundsError** exception if *index* lies beyond the end of the string.

– (void)**getCharacters:**(unichar *)*buffer*

Invokes **getCharacters:range:** with the provided *buffer* and the entire extent of the receiver as the range.

– (void)**getCharacters:**(unichar *)*buffer* **range:**(NSRange)*aRange*

Copies characters from *aRange* in the receiver into *buffer*, which must be large enough to contain them. This method does *not* add a null character. This method raises an **NSStringBoundsError** exception if any part of *aRange* lies beyond the end of the string.

Combining Strings

– (NSString *)**stringByAppendingFormat:**(NSString *)*format*,...

Returns a string made by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string.

– (NSString *)**stringByAppendingString:**(NSString *)*aString*

Returns a string made by appending *aString* and the receiver.

Dividing Strings into Substrings

– (NSArray *)**componentsSeparatedByString:**(NSString *)*separator*

Finds the substrings in the receiver that are delimited by *separator* and returns them as the elements of an NSArray. The strings in the array appear in the order they did in the receiver.

– (NSString *)**substringFromIndex:**(unsigned int)*index*

Returns a string object containing the characters of the receiver starting from the one at *index* to the end. This method raises an **NSStringBoundsError** exception if *index* lies beyond the end of the string.

– (NSString *)**substringFromRange:**(NSRange)*aRange*

Returns a string object containing the characters of the receiver which lie within *aRange*. This method raises an **NSStringBoundsError** exception if any part of *aRange* lies beyond the end of the string.

– (NSString *)**substringToIndex:**(unsigned int)*index*

Returns a string object containing the characters of the receiver up to, but not including, the one at *index*. This method raises an **NSStringBoundsError** exception if *index* lies beyond the end of the string.

Finding Ranges of Characters and Substrings

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*

Invokes **rangeOfCharacterFromSet:options:** with no options.

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*
options:(unsigned int)*mask*

Invokes **rangeOfCharacterFromSet:options:range:** with *mask* and the entire extent of the receiver as the range.

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns the range of the first character found from *aSet*. The search is restricted to *aRange* with *mask* options. *mask* can be any combination (using the C bitwise OR operator |) of NSCaseInsensitiveSearch, NSLiteralSearch, and NSBackwardsSearch.

– (NSRange)**rangeOfString:**(NSString *)*string*

Invokes **rangeOfString:options:** with no options.

– (NSRange)**rangeOfString:**(NSString *)*string*
options:(unsigned int)*mask*

Invokes **rangeOfString:options:range:** with *mask* options and the entire extent of the receiver as the range.

– (NSRange)**rangeOfString:**(NSString *)*aString*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns the range giving the location and length in the receiver of *aString*. The search is restricted to *aRange* with *mask* options. *mask* can be any combination (using the C bitwise OR operator |) of NSCaseInsensitiveSearch, NSLiteralSearch, NSBackwardsSearch, and NSAnchoredSearch.

Determining Composed Character Sequences

– (NSRange)**rangeOfComposedCharacterSequenceAtIndex:**(unsigned int)*anIndex*

Returns an NSRange giving the location and length in the receiver of the composed character sequence located at *anIndex*. This method raises an **NSStringBoundsError** exception if *anIndex* lies beyond the end of the string.

Identifying and Comparing Strings

– (NSComparisonResult)**caseInsensitiveCompare:**(NSString *)*aString*

Invokes **compare:options:** with the option NSCaseInsensitiveSearch.

– (NSComparisonResult)**compare:**(NSString *)*aString*

Invokes **compare:options:** with no options.

– (NSComparisonResult)**compare:**(NSString *)*aString*
options:(unsigned int)*mask*

Invokes **compare:options:range:** with *mask* as the options and the receiver's full extent as the range.

– (NSComparisonResult)**compare:**(NSString *)*aString*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Compares *aString* to the receiver and returns their lexical ordering. The comparison is restricted to *aRange* and uses *mask* options, which may be NSCaseInsensitiveSearch and NSLiteralSearch.

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

Returns YES if *aString* matches the beginning characters of the receiver, NO otherwise.

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

Returns YES if *aString* matches the ending characters of the receiver, NO otherwise.

– (unsigned int)**hash**

Returns an unsigned integer that can be used as a table address in a hash table structure. If two string objects are equal (as determined by the **isEqual:** method), they must have the same hash value**.**

– (BOOL)**isEqual:**(id)*anObject*

Returns YES if both the receiver and *anObject* have the same **id** or if they're both NSStrings that compare as **NSOrderedSame**, NO otherwise.

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

Returns YES if *aString* is equivalent to the receiver (if they have the same **id** or if they compare as **NSOrderedSame**), NO otherwise.

Storing the String

– (NSString *)**description**

Returns the string itself.

– (BOOL)**writeToFile:**(NSString *)*filename*
atomically:(BOOL)*useAuxiliaryFile*

Writes a textual description of the receiver to *filename.* If *useAuxiliaryFile* is YES, the data is written to a backup file and then, assuming no errors occur, the backup file is renamed to the intended file name. The string is written in the default C string encoding if the contents can be converted to that encoding. If not, the string is stored in the Unicode encoding.

Getting a Shared Prefix

– (NSString *)**commonPrefixWithString:**(NSString *)*aString*
options:(unsigned int)*mask*

Returns the substring of the receiver containing characters that the receiver and *aString* have in common. *mask* can be any combination (using the C bitwise OR operator |) of NSCaseInsensitiveSearch and NSLiteralSearch.

Changing Case

– (NSString *)**capitalizedString**

Returns a string with the first character of each word changed to its corresponding uppercase value.

– (NSString *)**lowercaseString**

Returns a string with each character changed to its corresponding lowercase value.

– (NSString *)**uppercaseString**

Returns a string with each character changed to its corresponding uppercase value.

Getting C Strings

– (const char *)**cString**

Returns a representation of the receiver as a C string in the default C string encoding.

– (unsigned int)**cStringLength**

Returns the length in bytes of the C string representation of the receiver.

– (void)**getCString:**(char *)*buffer*

Invokes **getCString:maxLength:range:remainingRange:** with NSMaximumStringLength as the maximum length, the receiver's entire extent as the range, and NULL for the remaining range. *buffer* must be large enough to contain the resulting C string plus a terminating null character (which this method adds).

– (void)**getCString:**(char *)*buffer*
maxLength:(unsigned int)*maxLength*

Invokes **getCString:maxLength:range:remainingRange:** with *maxLength* as the maximum length, the receiver's entire extent as the range, and NULL for the remaining range. *buffer* must be large enough to contain the resulting C string plus a terminating null character (which this method adds).

– (void)**getCString:**(char *)*buffer*
maxLength:(unsigned int)*maxLength*
range:(NSRange)*aRange*
remainingRange:(NSRange *)*leftoverRange*

Copies the receiver's characters (in the default C string encoding) as bytes into *buffer. buffer* must be large enough to contain *maxLength* bytes plus a terminating null character (which this method adds). Characters are copied from *aRange*; if not all characters can be copied, the range of those not copied is put into *leftoverRange*. This method raises an **NSStringBoundsError** exception if any part of *aRange* lies beyond the end of the string.

Getting Numeric Values

– (double)**doubleValue**

Returns the double precision floating point value of the receiver's text. Whitespace at the beginning of the string is skipped. If the receiver begins with a valid text representation of a floating-point number, that number's value is returned, otherwise 0.0 is returned. HUGE_VAL or –HUGE_VAL is returned on overflow. 0.0 is returned on underflow. Characters following the number are ignored.

– (float)**floatValue**

Returns the floating-point value of the receiver's text. Whitespace at the beginning of the string is skipped. If the receiver begins with a valid text representation of a floating-point number, that number's value is returned, otherwise 0.0 is returned. HUGE_VAL or –HUGE_VAL is returned on overflow. 0.0 is returned on underflow. Characters following the number are ignored.

– (int)**intValue**

Returns the integer value of the receiver's text. Whitespace at the beginning of the string is skipped. If the receiver begins with a valid representation of an integer, that number's value is returned, otherwise 0 is returned. INT_MAX or INT_MIN is returned on overflow. Characters following the number are ignored.

Working With Encodings

+ (NSStringEncoding *)**availableStringEncodings**

Returns a null terminated array of available string encodings..

+ (NSStringEncoding)**defaultCStringEncoding**

Returns the C string encoding assumed for any method accepting a C string as an argument.

+(NSString *)**localizedNameOfStringEncoding:**(NSStringEncoding)*encoding*

Returns the localized name of the string encoding specified by *encoding*.

– (BOOL)**canBeConvertedToEncoding:**(NSStringEncoding)*encoding*

Returns YES if the receiver can be converted to *encoding* without loss of information, and NO otherwise.

– (NSData *)**dataUsingEncoding:**(NSStringEncoding)*encoding*

Invokes **dataUsingEncoding:allowLossyConversion:** with NO as the argument to allow lossy conversion.

– (NSData *)**dataUsingEncoding:**(NSStringEncoding)*encoding* **allowLossyConversion:**(BOOL)*flag*

Returns an NSData object containing a representation of the receiver in *encoding*. If *flag* is NO and the receiver can't be converted without losing some information (such as accents or case) this method returns **nil**. If *flag* is YES and the receiver can't be converted without losing some information, some characters may be removed or altered in conversion.

– (NSStringEncoding)**fastestEncoding**

Encoding in which this string can be expressed (with lossless conversion) most quickly.

– (NSStringEncoding)**smallestEncoding**

Encoding in which this string can be expressed (with lossless conversion) in the most space efficient manner

Converting String Contents into a Property List

– (id)**propertyList**

Depending on the format of the receiver's contents, returns a string, data, array, or dictionary object represention of those contents.

– (NSDictionary *)**propertyListFromStringsFileFormat**

Returns a dictionary object initialized with the keys and values found in the receiver. The receiver's format must be that used for ".string" files.

Manipulating File System Paths

– (unsigned int)**completePathIntoString:**(NSString **)*outputName*
caseSensitive:(BOOL)*flag*
matchesIntoArray:(NSArray **)*outputArray*
filterTypes:(NSArray *)*filterTypes*

Regards the receiver as containing a partial filename and returns in *outputName* the longest matching path name. Case is considered if *flag* is YES. If *outputArray* is given, all matching filenames are return in *outputArray*. If *filterTypes* is provided, this method considers only those paths that match one of the types. Returns 0 if no matches are found; otherwise, the return value is positive.

– (NSString *)**lastPathComponent**

Returns the last component of the receiver's path representation. Given the path "/Images/Bloggs.tiff", this method returns a string containing "Bloggs.tiff".

– (NSString *)**pathExtension**

Returns the extension of the receiver's path representation. Given the path "/Images/Bloggs.tiff", this method returns a string containing "tiff".

– (NSString *)**stringByAbbreviatingWithTildeInPath**

Returns a string in which the user's home directory path is replace by "~".

– (NSString *)**stringByAppendingPathComponent:**(NSString *)*aString*

Returns a string representing the receiver's path with the addition of the path component *aString*.

– (NSString *)**stringByAppendingPathExtension:**(NSString *)*aString*

Returns a string representing the receiver's path with the addition of the extension *aString*.

– (NSString *)**stringByDeletingLastPathComponent**

Returns the receiver's path representation minus the last component. Given the path "/Images/Bloggs.tiff", this method returns a string containing "/Images".

– (NSString *)**stringByDeletingPathExtension**

Returns the receiver's path representation minus the extension on the last component. Given the path "/Images/Bloggs.tiff", this method returns a string containing "/Images/Bloggs".

– (NSString *)**stringByExpandingTildeInPath**

Returns a string in which a tilde is expanded to its full path equivalent.

– (NSString *)**stringByResolvingSymlinksInPath**

Returns a string identical to the receiver's path except that any symbolic links have been resolved.

– (NSString *)**stringByStandardizingPath**

Returns a string containing a "standardized" path, one in which tildes are expanded and redundant elements (for example "//") eliminated.

NSThread

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSThread.h

Class Description

An NSThread object controls a thread of execution. You use an NSThread when you want to terminate or delay a thread or you want a new thread.

A *thread* is an executable unit. A *task* is made up of one or more threads. Each thread has its own execution stack and is capable of independent I/O. All threads share the virtual memory address space and communication rights of their task. When a thread is started, it is *detached* from its initiating thread. The new thread runs independently. That is, the initiating thread does not know the new thread's state.

To obtain an NSThread object that represents your current thread of execution, use the **currentThread** method. To obtain an NSThread object that will create a new thread of execution, use **detachNewThreadSelector:toTarget:withObject:**. This method sends the specified Objective C message to the specified object in its own thread of execution. You use the NSThread object returned by these methods if you ever need to delay or terminate that thread of execution.

When you use **detachNewThreadSelector:toTarget:withObject:**, your application becomes multithreaded. At any time, you can send **isMultiThreaded** to find out if the application is multithreaded, that is, if a thread was ever detached from the current thread. **isMultiThreaded** returns YES even if the detached thread has completed execution.

Creating an NSThread

+ (NSThread *)**currentThread**	Returns an object representing the current thread of execution.
+ (void)**detachNewThreadSelector:**(SEL)*aSelector* **toTarget:**(id)*aTarget* **withObject:**(id)*anArgument*	Creates and starts a new NSThread for the message **[aTarget aSelector:anArgument]**. The method *aSelector* may take only one argument and may not have a return value. If this is the first thread detached from the current thread, this method posts the notification NSBecomingMultiThreaded will the nil object to the default notification center.

Querying a Thread

+ (BOOL)**isMultiThreaded**

Returns YES if a thread was ever detached (regardless of if the detached thread is still running).

– (NSMutableDictionary *)**threadDictionary**

Returns the NSThread's dictionary, allowing you to add data specific to the receiving NSThread. This essentially allows user-defined NSThread variables.

Delaying a Thread

+ (void)**sleepUntilDate:**(NSDate *)*date*

Has the receiving NSThread sleep until the time specified by *date*. No input or timers will be processed in this interval.

Terminating a Thread

+ (void)**exit**

Terminates the thread represented by the calling object. Before exiting that thread, this method posts the NSThreadExiting notification with the thread being exited to the default notification center.

NSTimer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSTimer.h

Class Description

NSTimer creates timer objects. A timer object waits until a certain time interval has elapsed and then fires, sending a specified message to a specified object. For example, you could create an NSTimer that sends a message to a window, telling it to update itself, after a certain time interval.

NSTimer objects work in conjunction with NSRunLoop objects. NSRunLoops control loops that wait for input, and they use NSTimers to help determine the maximum amount of time they should wait. When the NSTimer's time limit has elapsed, the NSRunLoop fires the NSTimer (causing its message to be sent), then checks for new input.

There are several ways to create an NSTimer object. The **scheduledTimerWithTimeInterval**... class methods automatically register the new NSTimer with the current NSRunLoop object in default mode. The **timerWithTimeInterval**... class methods create NSTimers that the user may register at a later time by sending the message **addTimer:forMode:** to the NSRunLoop. If you specify that the NSTimer should repeat, it will automatically reschedule itself after it fires. If a delay occurs when a timer is scheduled to fire, the timer will not fire. For example, suppose you used the following statement to create a timer:

```
timer = [NSTimer scheduledTimerWithTimeInterval:0.5 invocation:anInvocation repeats:YES];
```

This statement creates a timer will schedule itself to fire after 0.5 seconds, 1 second, 1.5 seconds, and so on from the time this statement is executed. Suppose there was a 2 second delay because NSRunLoop was busy processing input. The timer takes this delay into consideration and will skip intervals that were already missed when computing the next scheduled fire date.

There is no method that removes the association of an NSTimer from an NSRunLoop—send the NSTimer the **invalidate** message instead. **invalidate** disables the NSTimer, so it will no longer affect the NSRunLoop.

See the NSRunLoop class description for more information on NSRunLoops.

As a consequence of being a subclass of NSObject, NSTimer conforms to the NSCoding protocol. In practice, however, NSTimers are not encoded nor archived.

Creating a Timer Object

+ (NSTimer *)**scheduledTimerWithTimeInterval:**(NSTimeInterval)*seconds*
invocation:(NSInvocation *)*anInvocation*
repeats:(BOOL)*repeats*

Returns a new NSTimer object and registers it with the current NSRunLoop in the default mode. After *seconds* seconds have elapsed, the NSTimer fires, sending *anInvocation*'s message to its target. If *repeats* is YES, the NSTimer will repeatedly reschedule itself.

+ (NSTimer *)**scheduledTimerWithTimeInterval:**(NSTimeInterval)*seconds*
target:(id)*anObject*
selector:(SEL)*aSelector*
userInfo:(id)*anArgument*
repeats:(BOOL)*repeats*

Returns a new NSTimer object and registers it with the current NSRunLoop in the default mode. After *seconds* seconds have elapsed, the NSTimer fires, sending the message **[anObject aSelector:self]**. If *anObject* needs more information, it can send the NSTimer a **userData** message to retrieve *anArgument*. If *repeats* is YES, the NSTimer will repeatedly reschedule itself.

+ (NSTimer *)**timerWithTimeInterval:**(NSTimeInterval)*seconds*
invocation:(NSInvocation *)*anInvocation*
repeats:(BOOL)*repeats*

Returns a new NSTimer that, if registered, will fire after *seconds* seconds. Upon firing, the NSTimer sends *anInvocation*'s message to its target. If *repeats* is YES, the NSTimer will repeatedly reschedule itself.

+ (NSTimer *)**timerWithTimeInterval:**(NSTimeInterval)*seconds*
target:(id)*anObject*
selector:(SEL)*aSelector*
userInfo:(id)*anArgument*
repeats:(BOOL)*repeats*

Returns a new NSTimer that, if registered, will fire after *seconds* seconds. Upon firing, the NSTimer sends the message **[anObject aSelector:self]**. If *anObject* needs more information, it can send the NSTimer a **userData** message to retrieve *anArgument*. If *repeats* is YES, the NSTimer will repeatedly reschedule itself.

Firing the Timer

– (void)**fire**

Causes the NSTimer's message to be dispatched to its target.

Stopping the Timer

– (void)**invalidate**

Stops the NSTimer from ever firing again.

Getting Information About the NST imer

– (NSDate *)**fireDate**

Returns the date that the NSTimer will next fire.

– **userInfo**

Additional data that the object receiving NSTimer's message can use.

NSTimeZone

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: Foundation/NSDate.h

Class Description

NSTimeZone is an abstract class that defines the behavior of time-zone objects. By itself, NSDate represents dates as *universal time*. Universal time treats a date and time value as identical in, for instance, Redwood City and New York City. NSDate has no provision for locale adjustment of time-zone information. Provision for locale is critical for string descriptions and other expressions of conventional dates and times. NSTimeZone is used to affect the apparent value of date objects so that they reflect time zone related locale information.

NSTimeZoneDetail, a public subclass of NSTimeZone, augments the behavior of NSTimeZone by providing the commonly known attributes of a time zone in effect for a date within a time zone geopolitical area. These attributes are abbreviation, the offset from GMT (universal time), and an indication of whether Daylight Savings Time is in effect.

Time-zone objects represent geopolitical regions and use names to denote the various regions. For example, “US/Pacific” identifies the geopolitical time zone for San Francisco and Los Angeles, which falls in the same general latitude as that for the time zone “Canada/Pacific.” The US/Pacific time-zone has specific NSTimeZoneDetail instances that specify PST (Pacific Standard Time) and PDT (Pacific Daylight Time), which have slightly different offsets from GMT.

You typically associate the objects returned by NSTimeZone (and, by extension, NSTimeZoneDetail) with date objects to affect their behavior. Time-zone objects can be of various types:

- time zones with hour and minute offsets from Greenwich Mean Time (GMT)
- time zones with a single abbreviation and offset
- time zones that vary according to Standard Time and Daylight Savings Time

The system should supply the various choices for time zones along with time-zone information. These choices should be restricted to subsets based on latitude. You can access these choices through the **timeZoneArray** class method. Another restriction is the choice of time zone available when there is an ambiguous abbreviation; these choices are available through the class method **abbreviationDictionary**. Despite these restrictions, you can obtain an NSTimeZone object from an arbitrary file through the class method **timeZoneWithName**.

Note: By itself, the NSTimeZone class only *names* a time zone. It does not associate an abbreviation or a temporal offset with a time zone; that is done by NSTimeZoneDetail. An instance of NSTimeZone, however, “knows” about the set of time-zone detail objects related to it.

NSTimeZone provides several class methods to get time-zone objects, with or without detail: **timeZoneWithName:**, **timeZoneWithAbbreviation:**, and **timeZoneForSecondsFromGMT:**. The class also permits you to set the default time zone used by your application for your locale (**setDefaultTimeZone:**) You can access this default time zone at any time by the **defaultTimeZone** method, and, with the **localTimeZone** class method, you can also get a relative time-zone object that will decode itself to become the default time zone for any locale in which it finds itself.

NSCalendarDate methods return date objects that are automatically bound with time-zone detail objects. These date objects use the functionality of NSTimeZone to adjust dates for the proper locale. Unless you specify otherwise, objects returned from NSCalendarDate are bound to the default time zone for the current locale. A useful instance method is **timeZoneDetailForDate:**, which returns a time-zone detail object associated with a specific date.

Creating and Initializing an NSTimeZone

+ (NSTimeZoneDetail *)**defaultTimeZone**

Returns the default time zone as set for the current locale.

+ (NSTimeZone *)**localTimeZone**

Returns an NSTimeZone that behaves as the current default time zone in any given locale.

+ (NSTimeZone *)**timeZoneForSecondsFromGMT:**(int)*seconds*

Returns an NSTimeZone representing the time zone with *seconds* offset from Greenwich Mean Time. If there is no object matching the offset, this method creates and returns a new NSTimeZone bearing the value *seconds* as a name.

+ (NSTimeZoneDetail *)**timeZoneWithAbbreviation:**(NSString *)*abbreviation*

Returns the time-zone object identified by the abbreviation *abbreviation*. If there's no match, this method returns **nil**.

+ (NSTimeZone *)**timeZoneWithName:**(NSString *)*aTimeZoneName*

Returns the time-zone object with the name that corresponds to the geopolitical region *aTimeZoneName*. It searches the regions dictionary for matching names. If there is no match on the name, this method returns **nil**.

– (NSTimeZoneDetail *)**timeZoneDetailForDate:**(NSDate *)*date*

Returns the correct time-zone detail object associated with a date object. You invoke this method when a region's time zone (that is, its offset value from GMT) varies over the year, as happens between Standard Time and Daylight Savings Time.

Managing Time Zones

+ (void)**setDefaultTimeZone:**(NSTimeZone *)*aTimeZone*

Sets *aTimeZone* as the time zone appropriate for the current locale. This new time zone replaces the previous default time zone.

Getting Time Zone Information

+ (NSDictionary *)**abbreviationDictionary**

Returns a dictionary that maps abbreviations to region names, for example “PST” is the key for “US/Pacific”. If you know a region name for a key, you can obtain a valid abbreviation from the dictionary and use it to obtain a detail time-zone object using **timeZoneWithAbbreviation:**.

– (NSString *)**timeZoneName**

Returns the geopolitical name of the time zone.

Getting Arrays of Time Zones

+ (NSArray *)**timeZoneArray**

Returns an array of string object arrays, each containing strings that show current geopolitical names for each time zone. The subarrays are grouped by latitudinal region.

– (NSArray *)**timeZoneDetailArray**

Returns an array of NSTimeZoneDetail objects that are associated with the receiving NSTimeZone object.

NSTimeZoneDetail

Inherits From: NSTimeZone : NSObject

Conforms To: NSCoding, NSCopying (NSTimeZone)
NSObject (NSObject)

Declared In: Foundation/NSDate.h

Class Description

NSTimeZoneDetail is an abstract class that refines the behavior provided by NSTimeZone. NSTimeZone identifies a geopolitical area with a name (such as US/Pacific). NSTimeZoneDetail augments this region name with more specific information appropriate for a particular date within its geopolitical region: an abbreviation, an offset (in seconds) from Greenwich Mean Time (GMT), and an indication of whether Daylight Savings Time is in effect. The specificity afforded through NSTimeZoneDetail helps to resolve conflicts between abbreviations and offsets that can arise within regions.

Even though it is a concrete subclass of NSTimeZone, NSTimeZoneDetail does *not* have “factory” class methods that create and return time-zone objects. See the specification of NSTimeZone for methods that provide this ability.

However, NSTimeZoneDetail does have methods that allow you to get the abbreviation and temporal offset of a time-zone object, as well as determine whether Daylight Savings Time is in effect.

Querying an NSTimeZoneDetail

– (BOOL)**isDaylightSavingTimeZone** — Returns YES if the time-zone detail object is used in the representation of dates during Daylight Savings Time and returns NO otherwise.

– (NSString *)**timeZoneAbbreviation** — Returns the abbreviation of the time-zone detail object, such as EDT (Eastern Daylight Time).

– (int)**timeZoneSecondsFromGMT** — Returns the difference in seconds between the receiving time-zone detail object and Greenwich Mean Time. The offset can be a positive or negative value.

NSUnarchiver

Inherits From: NSCoder : NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSArchiver.h

Class Description

NSUnarchiver, a concrete subclass of NSCoder, defines objects that can decode a data structure, such as a graph of Objective C objects, from an archive. Such archives are produced by objects of the NSArchiver class. See the NSArchiver specification for an introduction to archiving.

General Exception Conditions

While unarchiving, NSUnarchiver performs a variety of consistency checks on the incoming data stream. NSUnarchiver raises an NSInconsistentArchiveException for a variety of reasons. Possible data errors leading to this exception are: unknown type descriptors in the data file; an array type descriptor is incorrectly terminated (missing]); excess characters in a type descriptor; a null class found where a concrete class was expected; class not loaded.

Initializing an NSUnarchiver

– (id)**initForReadingWithData:**(NSData *)*data*
Initializes an NSUnarchiver object from data object *data*. Raises **NSInvalidArgumentException** if the *data* argument is **nil**.

Decoding Objects

+ (id)**unarchiveObjectWithData:**(NSData *)*data*
Decodes an archived object stored in *data*.

+ (id)**unarchiveObjectWithFile:**(NSString *)*path*
Decodes an archived object stored in the file *path*.

– (void)**decodeArrayOfObjCType:**(const char *)*itemType*
count:(unsigned int)*count*
at:(void *)*array*
Decodes an *array* of *count* data elements of the same Objective C data *itemType*. It is your responsibility to release any objects derived in this way.

Managing an NSUnarchiver

– (BOOL)**isAtEnd**
Returns YES if the end of data is reached, NO if more data follows.

– (NSZone *)**objectZone** Returns the allocation zone for the unarchiver object.

– (void)**setObjectZone:**(NSZone *)*zone* Sets the allocation zone for the unarchiver object to *zone*. If *zone* is **nil**, it sets it to the default zone.

– (unsigned int)**systemVersion** Returns the system version number for the unarchived data.

Substituting One Class for Another

+ (NSString *)**classNameDecodedForArchiveClassName:**(NSString *)*nameInArchive*
Returns the class name used to archive instances of the class (*nameInArchive*). This may not be the original class name but another name encoded with NSArchiver's **encodeClassName:intoClassName**.

+ (void)**decodeClassName:**(NSString *)*nameInArchive*
asClassName:(NSString *)*trueName* Decodes from the archived data a class name (*nameInArchive*) substituted for the real class name (*trueName*). This method enables easy conversion of unarchived data when there are name changes in classes.

– (NSString *)**classNameDecodedForArchiveClassName:**(NSString *)*nameInArchive*
Returns the class name used to archive instances of the class (*nameInArchive*). This may not be the original class name but another name encoded with NSArchiver's **encodeClassName:intoClassName**.

– (void)**decodeClassName:**(NSString *)*nameInArchive*
asClassName:(NSString *)*trueName* Decodes from the archived data a class name (*nameInArchive*) substituted for the real class name (*trueName*). This method enables easy conversion of unarchived data when there are name changes in classes.

NSUserDefaults

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSUserDefaults.h

Class Description

The NSUserDefaults class allows an application to query and manipulate a user's defaults settings.

Defaults are grouped in domains. For example, there's a domain for application-specific defaults and another for global defaults. Each domain has a name and stores defaults as key-value pairs in an NSDictionary object. A default is identified by a string key, and its value can be any property-list object (NSData, NSString, NSArray, or NSDictionary). The standard domains are:

Domain	Identifier
Argument	NSArgumentDomain
Application	Identified by the application's name
Global	NSGlobalDomain
Languages	Identified by the language names
Registration	NSRegistrationDomain

The identifiers starting with "NS" above are global constants.

The argument domain is composed of defaults parsed from the application's arguments. The application domain contains the defaults set by the application. It's identified by the name of the application, as returned by this message:

```
NSString *applicationName = [[NSProcessInfo processInfo] processName];
```

The global domain contains defaults that are meant to be seen by all applications. The registration domain is a set of temporary defaults whose values can be set by the application to ensure that searches for default values will always be successful. Applications can create additional domains as needed.

A search for the value of a given default proceeds through the domains listed in an NSUserDefault object's *search list*. Only domains in the search list are searched. The standard search list contains the domains from the table above, in the order listed. A search ends when the default is found. Thus, if multiple domains contain the same default, only the domain nearest the beginning of the search list provides the default's value. Using the **searchList** method, you can reorder the default search list or set up one that is a subset of all the user's domains.

Typically, you use this class by invoking the **standardUserDefaults** class method to get an NSUserDefaults object. This method returns a global NSUserDefaults object with a search list already initialized. Then use the **setObject:forKey:** and **objectForKey:** methods to set and access user defaults.

The rest of the methods allow more complex defaults management. You can create your own domains, modify any domain, set up a custom search list, and even control the synchronization of the in-memory and on-disk defaults representations. The **synchronize** method saves any modifications to the persistent domains and updates all persistent domains that were not modified to what is on disk. **synchronize** is automatically invoked at periodic intervals.

You can create either persistent or volatile domains. Persistent domains are permanent and last past the life of the NSUserDefaults object. Any changes to the persistent domains are committed to disk. Volatile domains last only last as long as the NSUserDefaults object exists. The NSGlobalDomain domain is persistent; the NSArgumentDomain is volatile.

Warnings:

- User defaults are not thread safe.
- Automatic saving of changes to disk (through **synchronize**) depends on a run-loop being present.
- You should synchronize any domain you have altered before exiting a process.

Getting the Shared Instance

+ (NSUserDefaults *)**standardUserDefaults**

Returns the shared defaults object. If it doesn't exist yet, it's created with a search list containing the names of the following domains, in order: the NSArgumentDomain (consisting of defaults parsed from the application's arguments), a domain with the process' name, separate domains for each of the user's preferred languages, the NSGlobalDomain (consisting of defaults meant to be seen by all applications), and the NSRegistrationDomain (a set of temporary defaults whose values can be set by the application to ensure that searches will always be successful). The defaults are initialized for the current user. Subsequent modifications to the standard search list remain in effect even when this method is invoked again—the search list is guaranteed to be standard only the first time this method is invoked. The shared instance is provided as a convenience; other instances may also be created.

Getting and Setting a Default

– (NSArray *)**arrayForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSArray object (according to the **isKindOfClass:** test) and **nil** otherwise.

– (BOOL)**boolForKey:**(NSString *)*defaultName*

Invokes **stringForKey:** with key *defaultName*. Returns YES if the corresponding value is an NSString containing uppercase or lowercase "YES" or responds to the **intValue** message by returning a non-zero value. Otherwise, returns NO.

– (NSData *)**dataForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSData object (according to the **isKindOfClass:** test) and **nil** otherwise.

– (NSDictionary *)**dictionaryForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSDictionary object (according to the **isKindOfClass:** test) and **nil** otherwise.

– (float)**floatForKey:**(NSString *)*defaultName*

Invokes **stringForKey:** with key *defaultName*. Returns 0 if no string is returned. Otherwise, the resulting string is sent a **floatValue** message, which provides this method's return value.

– (int)**integerForKey:**(NSString *)*defaultName*

Invokes **stringForKey:** with key *defaultName*. Returns 0 if no string is returned. Otherwise, the resulting string is sent a **intValue** message, which provides this method's return value.

– (id)**objectForKey:**(NSString *)*defaultName*

Returns the value of the first occurrence of the specified default, searching the domains included in the search list. Returns **nil** if the default isn't found.

– (void)**removeObjectForKey:**(NSString *)*defaultName*

Removes the value for the given default in the standard application domain. Removing a default has no effect on the value returned by the **objectForKey:** method if the same key exists in a domain that precedes the standard application domain in the search list.

– (void)**setBool:**(BOOL)*value*
forKey:(NSString *)*defaultName*

Sets the value of the specified default to a string representation of YES or NO, depending on *value*. Invokes **setObject:forKey:** as part of its implementation.

– (void)**setFloat:**(float)*value*
forKey:(NSString *)*defaultName*

Sets the value of the specified default to a string representation of *value*. Invokes **setObject:forKey:** as part of its implementation.

– (void)**setInteger:**(int)*value*
forKey:(NSString *)*defaultName*

Sets the value of the specified default to a string representation of *value*. Invokes **setObject:forKey:** as part of its implementation.

– (void)**setObject:**(id)*value*
  forKey:(NSString *)*defaultName*

Sets the value of the specified default in the standard application domain. Setting a default has no effect on the value returned by the **objectForKey:** method if the same key exists in a domain that precedes the application domain in the search list.

– (NSArray *)**stringArrayForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSArray object containing NSStrings, and **nil** otherwise. The class of each object is determined using the **isKindOfClass:** test.

– (NSString *)**stringForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSString object (according to the **isKindOfClass:** test) and **nil** otherwise.

Initializing the User Defaults

– (id)**init**

Initializes defaults for the current user (who's identified by examining the environment). This method doesn't put anything in the search list. Invoke it only if you've allocated your own NSUserDefaults object instead of using the shared one. Returns **self**.

– (id)**initWithUser:**(NSString *)*userName*

Like **init**, but initializes defaults for the specified user.

Returning the Search List

– (NSMutableArray *)**searchList**

Returns a mutable array of domain names, signifying the domains that **objectForKey:** will search. You can customize the search list by modifying the array that's returned. Non-existent domain names in the list are ignored.

Maintaining Persistent Domains

– (NSDictionary *)**persistentDomainForName:**(NSString *)*domainName*

Returns a dictionary corresponding to the specified persistent domain. The keys in the dictionary are names of defaults, and the value corresponding to each key is a property list data object.

– (NSArray *)**persistentDomainNames**

Returns an array containing the names of the persistent domains. Each domain can then be retrieved by invoking **persistentDomainForName:**.

– (void)**removePersistentDomainForName:**(NSString *)*domainName*

Removes the named persistent domain from the user's defaults. The first time that a persistent domain is changed after **synchronize**, an NSUserDefaultsChanged notification is posted.

– (void)**setPersistentDomain:**(NSDictionary *)*domain* **forName:**(NSString *)*domainName*

Sets the dictionary for the persistent domain named *domainName*; raises an NSInvalidArgumentException if a volatile domain with *domainName* already exists. The first time that a persistent domain is changed after **synchronize**, an NSUserDefaultsChanged notification is posted.

– (BOOL)**synchronize**

Saves any modifications to the persistent domains and updates all persistent domains that were not modified to what is on disk. Returns NO if it could not save data to disk. Since the **synchronize** method is automatically invoked at periodic intervals, use this method only if you cannot wait for the automatic synchronization (for example if your application is about to exit), or if you want to update user defaults to what is on disk even though you have not made any changes.

Maintaining Volatile Domains

– (void)**removeVolatileDomainForName:**(NSString *)*domainName*

Removes the named volatile domain from the user's defaults.

– (void)**setVolatileDomain:**(NSDictionary *)*domain* **forName:**(NSString *)*domainName*

Sets the dictionary to *domain* for the volatile domain named *domainName*. This method raises an NSInvalidArgumentException if a persistent domain with *domainName* already exists.

– (NSDictionary *)**volatileDomainForName:**(NSString *)*domainName*

Returns a dictionary corresponding to the specified volatile domain. The keys in the dictionary are names of defaults, and the value corresponding to each key is a property list data object.

– (NSArray *)**volatileDomainNames**

Returns an array containing the names of the volatile domains. Each domain can then be retrieved by calling **volatileDomainForName:**.

Making Advanced Use of Defaults

– (NSDictionary *)**dictionaryRepresentation**

Returns a dictionary that contains a union of all key-value pairs in the domains in the search list. As with **objectForKey:**, key-value pairs in domains that are earlier in the search list take precedence. The combined result doesn't preserve information about which domain each entry came from.

– (void)**registerDefaults:**(NSDictionary *)*dictionary*

Adds the contents of *dictionary* to the registration domain. If there is no registration domain yet, it's created using *dictionary*, and NSRegistrationDomain is added to the end of the search list.

NSValue

Inherits From: NSObject

Conforms To: NSCoding, NSCopying
NSObject (NSObject)

Declared In: Foundation/NSValue.h
Foundation/NSGeometry.h

Class Description

NSValue objects provide an object-oriented wrapper for the data types defined in standard C and Objective C. The NSValue class is often used to put Objective C and standard C data types into collections that require objects, such as NSArray objects. When a value object is instantiated, it is encoded with the specified data type.

The NSValue class declares the programmatic interface to an object that contains a C data type. It provides methods for creating value objects that contain values of a specified data type, pointers, and other objects.

Use NSValue objects to put C types into collections. Use NSNumber objects to put numbers into collections.

The following code puts an NSRange into an NSArray, using the Objective C **@encode** directive to get a character string that encodes the type structure of NSRange:

```
[myArray insertObject:[NSValue value:&range withObjCType:@encode(NSRange)] atIndex:n]
```

To get the value back, you would do this:

```
[[myArray objectAtIndex:n] getValue:&range]
```

NSValue objects are provided with generic coding and copying behavior. To subclass NSValue and preserve class when encoding or copying, override **classForCoder**, **initWithCoder:, encodeWithCoder:** (for encoding), and **copyWithZone:** (for copying).

General Exception Conditions

NSValue can raise NSInternalInconsistencyException in a variety of cases where an unkown Objective C type is found. In addition, NSValue's implementation of **encodeWithCoder:** can raise NSInvalidArgumentException if an attempt is made to encode **void**.

Allocating and Initializing Value Objects

+ (NSValue *)**value:**(const void *)*value* **withObjCType:**(const char *)*type*
Creates and returns a value object containing the value *value* of the Objective C type *type*.

+ (NSValue *)**valueWithNonretainedObject:** (id)*anObject*
Creates and returns a value object containing the object *anObject*, without retaining *anObject*. This is provided as a convenience method: the statement [NSValue valueWithNonretainedOject:*anObject*] is equivalent to the statement [NSValue value:&*anObject* withObjCType:@encode(void *)].

+ (NSValue *)**valueWithPointer:**(const void *)*pointer*
Creates and returns a value object that contains the specified pointer. This is provided as a convenience method: the statement [NSValue valueWithPointer:*pointer*] is equivalent to the statement [NSValue value:&*pointer* withObjCType:@encode(void *)].

Allocating and Initializing Geometry Value Objects

+ (NSValue *)**valueWithPoint:**(NSPoint)*point*
Creates and returns a value object that contains the specified NSPoint structure (which represents a geometrical point in two dimensions).

+ (NSValue *)**valueWithRect:**(NSRect)*rect*
Creates and returns a value object that contains the specified NSRect structure, representing a rectangle.

+ (NSValue *)**valueWithSize:**(NSSize)*size*
Creates and returns a value object that contains the specified NSSize structure (which stores a width and a height).

Accessing Data in Value Objects

– (void)**getValue:**(void *)*value*
Copies the receiver's data into *value*.

– (id)**nonretainedObjectValue**
Returns the non-retained object that's contained in the receiver. It's an error to send this message to an NSValue object that doesn't store a nonretained object.

– (const char *)**objCType**
Returns the Objective C type of the data contained in the receiver.

– (void *)**pointerValue**
Returns the value pointed to by a pointer contained in an value object. It's an error to send this message to an NSValue that doesn't store a pointer.

Accessing Data in Value Geometry Objects

– (NSPoint)**pointValue**	Returns the point structure that's contained in the receiver.
– (NSRect)**rectValue**	Returns the rectangle structure that's contained in the receiver.
– (NSSize)**sizeValue**	Returns the size structure that's contained in the receiver.

Protocols

NSCoding

Adopted By: NSObject

Declared In: Foundation/NSObject.h

Protocol Description

The NSCoding protocol declares the two methods that a class must implement so that objects of that class can be encoded and decoded. This capability provides the basis for archiving (where objects and other structures are stored on disk) and distribution (where objects are copied to different address spaces).

When an object receives an **encodeWithCoder:** message, it should write its instance variables (and, through a message to **super**, the instance variables that it inherits) to the supplied NSCoder. Similarly, when an object receives an **initWithCoder:** message, it should initialize its instance variables (and inherited instance variables, again through a message to **super**) from the data in the supplied NSCoder. See the NSCoder and NSArchiver class specifications for more complete information.

Encoding and Decoding Objects

– (void)**encodeWithCoder:**(NSCoder *)*aCoder*	Encodes the receiver using *aCoder.*
– (id)**initWithCoder:**(NSCoder *)*aDecoder*	Initializes and returns a new instance from data in *aDecoder.*

NSCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

A class whose instances provide functional copies of themselves should adopt the NSCopying protocol. The exact meaning of “copy” can vary from class to class, but a copy must be a functionally independent object, identical to the original at the time the copy was made. Where the concept “immutable vs. mutable” applies to an object, this protocol produces immutable copies; see the NSMutableCopying protocol for details on making mutable copies. Property list classes (NSString, NSData, NSArray, and NSDictionary) guarantee immutable returned values.

In most cases, to produce a copy that’s independent of the original, a *deep copy* must be made. A deep copy is one in which every instance variable of the receiver is duplicated, instead of referencing the variable in the original object. If the receiver’s instance variables themselves have instance variables, those too must be duplicated, and so on. A deep copy is thus a completely separate object from the original; changes to it don’t affect the original, and changes to the original don’t affect it. Further, for an immutable copy, no part at any level may be changed, making a copy a “snapshot” of the original object.

Making a complete deep copy isn’t always needed. Some objects can reasonably share instance variables among themselves—a static string object that gets replaced but not modified, for example. In such cases your class can implement NSCopying more cheaply than it might otherwise need to.

The typical usage of NSCopying is to create “passing by value” value objects.

Contrary to most methods, the returned object is owned by the caller, who is responsible for releasing it.

Copying Objects

– (id)**copyWithZone:**(NSZone *)*zone*

Returns a new instance that’s a functional copy of the receiver. Memory for the new instance is allocated from zone. For collections, creates a deep (recursive) copy. The copy returned is immutable if the consideration “immutable vs. mutable” applies to the receiving object; otherwise the exact nature of the copy is determined by the class. The returned object is owned by the caller, who is responsible for releasing it.

NSLocking

Adopted By: NSConditionLock
NSLock
NSRecursiveLock

Declared In: Foundation/NSLock.h

Protocol Description

This protocol is used by classes that provide lock objects. The lock objects provided by OpenStep are used only for protecting critical sections of code: sections that manipulate shared data and that can be executed simultaneously in several threads. Lock objects—except for NSConditionLock objects—contain no useful data.

Although an object that isn't a lock could adopt the NSLocking protocol, it may be more desirable to design the object so that all locking is handled internally, through normal use rather than requiring that the object be explicitly locked and unlocked.

In order to enable clients to only have locks when processes become multithreaded, it is permissible to unlock a lock freshly created (i.e. that has not been locked)—unless it is a recursive lock.

Three classes conform to the NSLocking protocol:

Class	Usage
NSLock	Protect critical sections of code.
NSConditionLock	Protects critical sections of code, but can also be used to postpone entry to a critical section until a condition is met. This class is functionally a superset of the NSLock class, though unlocking is slightly more expensive.
NSRecursiveLock	Protects critical sections from access by multiple threads, but allows a single thread to acquire a lock several times without deadlocking.

None of these classes busy-waits while the lock is unavailable. All classes may all be efficiently used for long sections of atomic code. See the class specifications for these classes for further information on their behavior and usage.

Locking Operations

– (void)**lock**

Acquires a lock. Applications generally do this when entering a critical section of their code. A thread will sleep if it can't immediately acquire the lock.

– (void)**unlock**

Releases a lock. Applications generally do this when exiting a critical section of their code.

NSMutableCopying

Adopted By: various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

A class that defines an “immutable vs. mutable” distinction adopts this protocol to allow mutable copies of its instances to be made. A mutable copy of an object is usually a *shallow copy* (as opposed to the *deep copy* defined in the NSCopying protocol specification). The original and its copy share references to the same instance variables, so that if a component of the copy is changed, for example, that change is reflected in the original.

A class that doesn’t define an “immutable vs. mutable” distinction but that needs to offer both deep and shallow copying shouldn’t adopt this protocol. The NSCopying methods should by default be assumed to produce deep copies; the class can then also implement methods to produce shallow copies.

Contrary to most methods, the returned value is owned by the caller, who is responsible for releasing it.

Making Mutable Copies of Objects

– (id)**mutableCopyWithZone:**(NSZone *)*zone*

Returns a new instance that’s a top level, mutable copy of the receiver. For a collection, objects in the collection are retained. Memory for the new instance is allocated from *zone*. The returned object is owned by the caller, who is responsible for releasing it.

NSObjCTypeSerializationCallBack

Adopted By: No OpenStep classes

Declared In: Foundation/NSSerialization.h

Protocol Description

An object conforms to the NSObjCTypeSerializationCallBack protocol so that it can intervene in the serialization and deserialization process. The primary purpose of this protocol is to allow for the serialization of objects and other data types that aren't directly supported by OpenStep's serialization facility. (See the NSSerializer class specification for information on serialization.)

NSMutableData declares the method that's used to begin the serialization process:

```
- (void)serializeDataAt:(const void *)data
  ofObjCType:(const char *)type
  context:(id <NSObjCTypeSerializationCallBack>)callback
```

This method can serialized all standard Objective C types (**int**, **float**, character strings, and so on) except for objects, **union**, and **void ***. If, during the serialization process, an object is encountered, the object passed as the callback argument above is asked to provide the serialization.

Suppose that the type being serialized is a structure of this description:

```
struct stockRecord {
  NSString *stockName;
  float value;
};
```

The Objective C type code for this structure is {@f}, so the serialization process begins with this message: (Assume that **theData** is the NSMutableData object that's doing the serialization and **helper** is an object that conforms to the NSObjCTypeSerializationCallBack protocol.)

```
struct stockRecord aRecord = {@"aCompany", 34.7};

[theData serializeDataAt:&aRecord ofObjCType:"{@f}" context:helper];
```

Since the first field of the structure is an unsupported type, the helper object is sent a **serializeObjectAt:ofObjCType:intoData:** message, letting it serialize the object. **helper** might implement the method in this way:

```
- (void)serializeObjectAt:(id *)objectPtr
    ofObjCType:(const char *)type
    intoData:(NSMutableData *)theMutableData
{
    NSString *nameObject;
    char *companyName

    nameObject = *objectPtr;
    companyName = [nameObject cString];

    [theData serializeDataAt:&companyName ofObjCType:@encode(typeof(companyName))
        context:nil]
}
```

The callback object is free to serialize the target object as it wishes. In this case, **helper** simply extracts the company name from the NSString object and then has that character string serialized. Once this callback method finishes executing, the original method (**serializeDataAt:ofObjCType:context:**) resumes execution and serializes the second field of the structure. Since this second field contains a supported type (**float**), the callback method is not invoked again.

Deserialization follows a similar pattern, except in this case NSData declares the central method **deserializeDataAt:ofObjCType:atCursor:context:**. The deserialization of the example structure starts with a message to the NSData object that contains the serialized data:

```
(unsigned *)cursor = 0;

[theData deserializeDataAt:&aRecord ofObjCType:"{@f}" cursor:&cursor context:helper];
```

(The cursor argument is a pointer to zero since we're starting at the beginning of the data in the NSData object.)

When this method is invoked, the callback object receives a **deserializeObjectAt:ofObjCType:fromData:atCursor:** message, as declared in this protocol. The callback object can then reestablish the first field of the structure. For example, **helper** might implement the method in this way:

```
- (void) deserializeObjectAt:(id *)objectPtr
  ofObjCType:(const char *)type
  fromData:(NSData *)data
  atCursor:(unsigned *)cursor
{
  char *companyName;

  [theData deserializeDataAt:&companyName ofObjCType:"*" atCursor:cursor context:nil];
  *objectPtr = [[NSString stringWithCString:companyName] retain];
}
```

Callback Handling

– (void)**deserializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
fromData:(NSData *)*data*
atCursor:(unsigned int*)*cursor*

The implementor of this method decodes the referenced *object* (which should always be of *type* "@") located at the *cursor* position in the *data* object. The decoded object is *not* autoreleased. See description of NSData method **deserializeDataAt:ofObjCType:context:**.

– (void)**serializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
intoData:(NSMutableData *)*data*

The implementor of this method encodes the referenced *object* (which should always be of *type* "@") in the *data* object. See description of NSMutableData method **serializeDataAt:ofObjCType:context:**

NSObject

Adopted By: NSObject
NSProxy

Declared In: Foundation/NSObject.h

Protocol Description

The NSObject protocol declares methods that all objects—no matter which root class they descend from (NSObject, NSProxy, or another root class)—should implement to work well within OpenStep. Some of the methods in this protocol reveal an object's primary attributes: its position in the class hierarchy, its conformance to other protocols, and whether it responds to specific messages. Others let it be manipulated in various ways. For example, it can be asked to perform methods that are determined at runtime (using the **perform:...** methods) or to participate in OpenStep's automatic deallocation scheme (using the **retain**, **release**, and **autorelease** methods).

By conforming to this protocol an object advertises that it has the basic behaviors necessary to work with the OpenStep's container classes (such as NSArray or NSDictionary).

Identifying and Comparing Instances

– (unsigned int)**hash**	Returns an unsigned integer that can be used as a table address in a hash table structure. Two objects that are equal must hash to the same value.
– (BOOL)**isEqual:**(id)*anObject*	Returns YES if the receiver and *anObject* have equal values; otherwise returns NO.
– (id)**self**	Returns the receiver.

Identifying Class and Superclass

– (Class)**class**	Returns the class object for the receiver's class.
– (Class)**superclass**	Returns the class object for the receiver's superclass.

Determining Allocation Zones

– (NSZone *)**zone**	Returns a pointer to the zone from which the receiver was allocated.

Sending Messages Determined at Run Time

– (id)**perform:**(SEL)*aSelector*

Sends an *aSelector* message to the receiver and returns the result of the message. If *aSelector* is null, an NSInvalidArgumentException is raised.

– (id)**perform:**(SEL)*aSelector*
withObject:(id)*anObject*

Sends an *aSelector* message to the receiver with *anObject* as an argument. If *aSelector* is null, an NSInvalidArgumentException is raised.

– (id)**perform:**(SEL)*aSelector*
withObject:(id)*anObject*
withObject:(id)*anotherObject*

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. If *aSelector* is null, an NSInvalidArgumentException is raised.

Identifying Proxies

– (BOOL)**isProxy**

Returns YES to indicate that the receiver is an NSProxy, rather than an object that descends from NSObject. Otherwise, it returns NO.

Testing Inheritance Relationships

– (BOOL)**isKindOfClass:**(Class)*aClass*

Returns YES if the receiver is an instance of *aClass* or an instance of any class that inherits from *aClass*. Otherwise, it returns NO.

– (BOOL)**isMemberOfClass:**(Class)*aClass*

Returns YES if the receiver is an instance of *aClass*. Otherwise, it returns NO.

Testing for Protocol Conformance

– (BOOL)**conformsToProtocol:**(Protocol *)*aProtocol*

Returns YES if the class of the receiver conforms to *aProtocol*, and NO if it doesn't.

Testing Class Functionality

– (BOOL)**respondsToSelector:**(SEL)*aSelector*

Returns YES if the receiver implements or inherits a method that can respond to *aSelector* messages, and NO if it doesn't.

Managing Reference Counts

– (id)**autorelease**

As defined in the NSObject class, decrements the receiver's reference count. When the count reaches 0, adds the object to the current autorelease pool. Returns **self**. Objects in the pool are released later, typically at the top of the event loop.

– (oneway void)**release**

As defined in the NSObject class, decrements the receiver's reference count. When the count reaches 0, the object is automatically deallocated immediately.

– (id)**retain**

As defined in the NSObject class, **retain** increments the receiver's reference count. You send an object a **retain** message when you want to prevent it from being deallocated without your express permission. Returns **self** as a convenience.

– (unsigned int)**retainCount**

Returns the receiver's reference count for debugging purposes.

Describing the Object

– (NSString *)**description**

Returns a human-readable description of the receiver.

Foundation Kit Functions

Memory Allocation Functions

Get the Virtual Memory Page Size

unsigned **NSPageSize**(void) — Returns the number of bytes in a page.

unsigned **NSLogPageSize**(void) — Returns the binary log of the page size.

unsigned **NSRoundDownToMultipleOfPageSize**(unsigned *byteCount*)
Returns the multiple of the page size that is closest to, but not greater than, *byteCount*.

unsigned **NSRoundUpToMultipleOfPageSize**(unsigned *byteCount*)
Returns the multiple of the page size that is closest to, but not less than, *byteCount*.

Get the Amount of Real Memory

unsigned **NSRealMemoryAvailable**(void) — Returns the number of bytes available in the RAM.

Allocate or Free Virtual Memory

void ***NSAllocateMemoryPages**(unsigned *byteCount*)
Allocates the integral number of pages whose total size is closest to, but not less than, *byteCount*, with the pages guaranteed to be zero-filled.

void **NSDeallocateMemoryPages**(void **pointer*, unsigned *byteCount*) — Deallocates memory that was allocated with **NSAllocateMemoryPages()**.

void **NSCopyMemoryPages**(const void **source*, void **destination*, unsigned *byteCount*) — Copies (or copies-on-write) *byteCount* bytes from *source* to *destination*.

Get a zone

NSZone ***NSCreateZone**(unsigned *startSize*, unsigned *granularity*, BOOL *canFree*) — Creates and returns pointer to a new zone of *startSize* bytes, that grows and shrinks by *granularity* bytes. If *canFree* is NO, the allocator never frees memory, and **malloc()** will be fast.

NSZone ***NSDefaultMallocZone**(void)	Returns the default zone, which is created automatically at startup. This is the zone used by **malloc()**.
NSZone ***NSZoneFromPointer**(void **pointer*)	Returns the zone for the *pointer* block of memory, or NULL if the block wasn't allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

Allocate or Free Memory in a Zone

void ***NSZoneMalloc**(NSZone **zone*, unsigned *size*)	Allocates *size* bytes in *zone*, and returns a pointer to the allocated memory.
void ***NSZoneCalloc**(NSZone **zone*, unsigned *numElems*, unsigned *numBytes*)	Allocates memory from *zone* for *numElems* elements, each with a size of *numBytes*, and returns a pointer to the memory. The memory is initialized with zeros.
void ***NSZoneRealloc**(NSZone **zone*, void **pointer*, unsigned *size*)	Changes the size of the block of memory pointed to by *pointer* to *size* bytes. It may allocate new memory to replace the old, in which case it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes. The *pointer* may be NULL.
void **NSRecycleZone**(NSZone **zone*)	Frees *zone* after adding any of its pointers still in use to the default zone. (This strategy prevents retained objects from being inadvertently destroyed.)
void **NSZoneFree**(NSZone **zone*, void **pointer*)	Returns memory to the zone from which it was allocated. The standard C function **free()** does the same, but spends time finding which zone the memory belongs to.

Name a Zone

void **NSSetZoneName**(NSZone **zone*, NSString **name*)	Sets the specified zone's name to *name*, which can aid in debugging.
NSString ***NSZoneName**(NSZone **zone*)	Returns the name of *zone*.

Object Allocation Functions

Allocate or Free an Object

NSObject ***NSAllocateObject**(Class *aClass*,
unsigned *extraBytes*,
NSZone **zone*)

Allocates and returns a pointer to an instance of *aClass*, created in the specified zone (or in the default zone, if *zone* is NULL). *extraBytes* (usually 0) states the number of extra bytes required for indexed instance variables.

NSObject ***NSCopyObject**(NSObject **anObject*,
unsigned *extraBytes*,
NSZone **zone*)

Creates and returns a new object that's an exact copy of *anObject*. The second and third arguments have the same meaning as in **NSAllocateObject()**.

void **NSDeallocateObject**(NSObject **anObject*)

Deallocates *anObject*, which must have been allocated using **NSAllocateObject()**.

Decide Whether to Retain an Object

BOOL **NSShouldRetainWithZone**(NSObject **anObject*,
NSZone **requestedZone*)

Returns YES if *requestedZone* is NULL, the default zone, or the zone in which *anObject* was allocated. This function is typically called from inside an NSObject's **copyWithZone:** method, when deciding whether to retain *anObject* as opposed to making a copy of it.

Modify the Number of References to an Object

BOOL **NSDecrementExtraRefCountWasZero**(id *anObject*)

Returns YES if the externally maintained "extra reference count" for *anObject* is zero; otherwise, this function decrements the count and returns NO.

void **NSIncrementExtraRefCount**(id *anObject*)

Increments the externally maintained "extra reference count" for *anObject*. The first reference (typically done in the **+alloc** method) isn't maintained externally, so there's no need to call this function for that first reference.

Error-Handling Functions

Change the Top-level Error Handler

NSUncaughtExceptionHandler ***NSGetUncaughtExceptionHandler**(void)

Returns a pointer to the function serving as the top-level error handler. This handler will process exceptions raised outside of any exception-handling domain.

void **NSSetUncaughtExceptionHandler**(NSUncaughtExceptionHandler **handler*)

Sets the top-level error-handling function to *handler*. If *handler* is NULL or this function is never invoked, the default top-level handler is used.

Macros to Handle an Exception

NS_DURING

Marks the beginning of an exception-handling domain (a portion of code delimited by NS_DURING and NS_HANDLER). When an error is raised anywhere within the exception-handling domain, program execution jumps to the first line of code in the exception handler. It's illegal to exit the exception-handling domain by any other means than NS_VALUERETURN, NS_VOIDRETURN, or falling out the bottom.

NS_ENDHANDLER

Marks the ending of an exception handler (a portion of code delimited by NS_HANDLER and NS_ENDHANDLER).

NS_HANDLER

Marks the ending of an exception-handling domain and the beginning of the corresponding exception handler. Within the scope of the handler, a local variable called exception stores the raised exception. Code delimited by NS_HANDLER and NS_ENDHANDLER is never executed except when an error is raised in the preceding exception-handling domain.

value **NS_VALUERETURN(***value***,** *type***)**

Causes the method (or function) in which this macro occurs to immediately return *value* of type *type*. This macro can only be used within an exception-handling domain.

NS_VOIDRETURN

Causes the method (or function) in which this macro occurs to return immediately, with no return value. This macro can only be placed within an exception-handling domain.

Call the Assertion Handler from the Body of an Objective-C Method

NSAssert(BOOL *condition,*
NSString **description***)**

Calls the NSAssertionHandler object for the current thread if *condition* is false. The *description* should explain the error, formatted as for the standard C function **printf**(); it need not include the object's class and method name, since they're passed automatically to the handler.

NSAssert1(BOOL *condition*,
NSString **description*,
*arg***)**

Like **NSAssert()**, but the format string description includes a conversion specification (such as %s or %d) for the argument *arg*, in the style of **printf()**. You can pass an object in *arg* by specifying **%@**, which gets replaced by the string that the object's **description** method returns.

NSAssert2(BOOL *condition***,**
NSString **description***,**
arg1,
*arg2***)**

Like **NSAssert1()**, but with two arguments.

NSAssert3(BOOL *condition***,**
NSString **description***,**
*arg1***,**
*arg2***,**
*arg3***)**

Like **NSAssert1()**, but with three arguments.

NSAssert4(BOOL *condition*,
NSString **description*,
arg1,
arg2,
arg3,
*arg4***)**

Like **NSAssert1()**, but with four arguments.

NSAssert5(BOOL *condition*,
NSString **description*,
arg1,
arg2,
arg3,
arg4,
*arg5***)**

Like **NSAssert1()**, but with five arguments.

Call the Assertion Handler from the Body of a C Function

NSCAssert(BOOL *condition*,
NSString **description*)

Calls the NSAssertionHandler object for the current thread if *condition* is false. The *description* should explain the error, formatted as for the standard C function **printf()**; it need not include the function name, which is passed automatically to the handler.

NSCAssert1(BOOL *condition*,
NSString **description*,
arg)

Like **NSCAssert()**, but the format string *description* includes a conversion specification (such as **%s** or **%d**) for the argument *arg*, in the style of **printf()**.

NSCAssert2(BOOL *condition*,
NSString **description*,
arg1,
arg2)

Like **NSCAssert1()**, but with two arguments.

NSCAssert3(BOOL *condition*,
NSString **description*,
arg1,
arg2,
arg3)

Like **NSCAssert1()**, but with three arguments.

NSCAssert4(BOOL *condition*,
NSString **description*,
arg1,
arg2,
arg3,
arg4)

Like **NSCAssert1()**, but with four arguments.

NSCAssert5(BOOL *condition*,
NSString **description*,
arg1,
arg2,
arg3,
arg4,
arg5)

Like **NSCAssert1()**, but with five arguments.

Validate a Parameter

NSParameterAssert(BOOL *condition*)

Like **NSAssert()**, but the description passed is "Invalid parameter not satisfying: " followed by the text of *condition* (which can be any boolean expression).

NSCParameterAssert(BOOL *condition*)

Like **NSParameterAssert()**, but to be called from the body of a C function.

Geometric Functions

Create Basic Structures

NSPoint **NSMakePoint(**float *x*, float *y***)**

Create an NSPoint having the coordinates *x* and *y*.

NSSize **NSMakeSize(**float *w*, float *h***)** — Create an NSSize having the specified width and height.

NSRect **NSMakeRect(**float *x*, float *y*, float *w*, float *h***)** — Create an NSRect having the specified origin and size.

NSRange **NSMakeRange(**unsigned int *location*, unsigned int *length***)** — Create an NSRange having the specified location and length.

Get a Rectangle's Coordinates

float **NSMaxX**(NSRect *aRect*) — Returns the largest x-coordinate value within *aRect.*

float **NSMaxY**(NSRect *aRect*) — Returns the largest y-coordinate value within *aRect*.

float **NSMidX**(NSRect *aRect*) — Returns the x-coordinate of the rectangle's center point.

float **NSMidY**(NSRect *aRect*) — Returns the y-coordinate of the rectangle's center point.

float **NSMinX**(NSRect *aRect*) — Returns the smallest x-coordinate value within *aRect*.

float **NSMinY**(NSRect *aRect*) — Returns the smallest y-coordinate value within*aRect* .

float **NSWidth**(NSRect *aRect*) — Returns the width of *aRect*.

float **NSHeight**(NSRect *aRect*) — Returns the height of *aRect*.

Modify a Copy of a Rectangle

NSRect **NSInsetRect**(NSRect *aRect*,
float *dX*,
float *dY*)

Returns a copy of the rectangle *aRect*, altered by moving the two sides that are parallel to the y-axis inwards by *dX*, and the two sides parallel to the x-axis inwards by *dY.*

NSRect **NSOffsetRect**(NSRect *aRect*,
float *dX*,
float *dY*)

Returns a copy of the rectangle *aRect*, with its location shifted by *dX* along the x-axis and by *dY* along the y-axis.

void **NSDivideRect**(NSRect *inRect*,
NSRect **slice*,
NSRect **remainder*,
float *amount*,
NSRectEdge *edge*)

Creates two rectangles, *slice* and *remainder,* from *inRect*, by dividing *inRect* with a line that's parallel to one of *inRect*'s sides (namely, the side specified by edge—either NSMinXEdge, NSMinYEdge, NSMaxXEdge, or NSMaxYEdge). The size of *slice* is determined by *amount*, which measures the distance from *edge*.

NSRect **NSIntegralRect**(NSRect *aRect*) — Returns a copy of the rectangle *aRect*, expanded outwards just enough to ensure that none of its four defining values (*x*, *y*, *width*, and *height*) have fractional parts. If *aRect*'s *width* or *height* is zero or negative, this function returns a rectangle with origin at (0.0, 0.0) and with zero width and height.

Compute a Third Rectangle from Two Rectangles

NSRect **NSUnionRect**(NSRect *aRect*,
NSRect *bRect*)

Returns the smallest rectangle that completely encloses both *aRect* and *bRect*. If one of the rectangles has zero (or negative) width or height, a copy of the other rectangle is returned; but if both have zero (or negative) width or height, the returned rectangle has its origin at (0.0, 0.0) and has zero width and height.

NSRect **NSIntersectionRect**(NSRect *aRect*,
NSRect *bRect*)

Returns the graphic intersection of *aRect* and *bRect*. If the two rectangles don't overlap, the returned rectangle has its origin at (0.0, 0.0) and zero width and height. (This includes situations where the intersection is a point or a line segment.)

Test Geometric Relationships

BOOL **NSEqualRects**(NSRect *aRect*,
NSRect *bRect*)

Returns YES if the two rectangles *aRect* and *bRect* are identical, and NO otherwise.

BOOL **NSEqualSizes**(NSSize *aSize*,
NSSize *bSize*)

Returns YES if the two sizes *aSize* and *bSize* are identical, and NO otherwise.

BOOL **NSEqualPoints**(NSPoint *aPoint*,
NSPoint *bPoint*)

Returns YES if the two points *aPoint* and *bPoint* are identical, and NO otherwise.

BOOL **NSIsEmptyRect**(NSRect *aRect*)

Returns YES if the rectangle encloses no area at all—that is, if its width or height is zero or negative.

BOOL **NSMouseInRect**(NSPoint *aPoint*,
NSRect *aRect*,
BOOL *flipped*)

Returns YES if the point represented by *aPoint* is located within the rectangle represented by *aRect*. It assumes an unscaled and unrotated coordinate system; the argument *flipped* should be YES if the coordinate system has been flipped so that the positive y-axis extends downward. This function is used to determine whether the hot spot of the cursor lies inside a given rectangle.

BOOL **NSPointInRect**(NSPoint *aPoint*,
NSRect *aRect*)

Performs the same test as **NSMouseInRect()**, but assumes a flipped coordinate system.

BOOL **NSContainsRect**(NSRect *aRect*,
NSRect *bRect***)**

Returns YES if *aRect* completely encloses *bRect*. For this to be true, *bRect* can't be empty and none of its sides can touch any of *aRect*'s.

Get a String Representation

NSString ***NSStringFromPoint**(NSPoint *aPoint*)

Returns a string of the form "{x=*a*; y=*b*}", where *a* and *b* are the x- and y-coordinates of *aPoint*.

NSString ***NSStringFromRect**(NSRect *aRect*)

Returns a string of the form "{x=*a*; y=*b*; width=*c*; height=*d*}", where *a*, *b*, *c*, and *d* are the x- and y-coordinates and the width and height, respectively, of *aRect*.

NSString ***NSStringFromSize**(NSSize *aSize*)

Returns a string of the form "{width=*a*; height=*b*}", where *a* and *b* are the width and height of *aSize*.

Range Functions

Query a Range

BOOL **NSEqualRanges**(NSRange *range1*, NSRange *range2*)

Returns YES if *range1* and *range2* have the same locations and lengths.

unsigned **NSMaxRange**(NSRange *range*)

Returns *range*.location + *range*.length—in other words, the number one greater than the maximum value within the range.

BOOL **NSLocationInRange**(unsigned *location*, NSRange *range*)

Returns YES if *location* is in *range* (that is, if *location* is greater than or equal to *range*.location and *location* is less than **NSMaxRange**(*range*)).

Compute a Range from Two Other Ranges

NSRange **NSUnionRange**(NSRange *range1*, NSRange *range2*)

Returns a range whose maximum value is the greater of *range1*'s and *range2*'s maximum values, and whose location is the lesser of the two range's locations.

NSRange **NSIntersectionRange**(NSRange *range1*, NSRange *range2*)

Returns a range whose maximum value is the lesser of *range1*'s and *range2*'s maximum values, and whose location is the greater of the two range's locations. However, if the two ranges don't intersect, the returned range has a location and length of zero.

Get a String Representation

NSString ***NSStringFromRange**(NSRange *range*)

Returns a string of the form: "{location = *a*; length = *b*}", where *a* and *b* are non-negative integers.

Hash Table Functions

Create a Table

NSHashTable ***NSCreateHashTable**(NSHashTableCallBacks *callBacks*,
unsigned *capacity*)

Creates, and returns a pointer to, an NSHashTable in the default zone; the table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash()** is NULL; pointer equality, if **isEqual()** is NULL; no call-back upon adding an element, if **retain()** is NULL; no call-back upon removing an element, if **release()** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe()** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

NSHashTable ***NSCreateHashTableWithZone**(NSHashTableCallBacks *callBacks*,
unsigned *capacity*,
NSZone **zone*)

Like **NSCreateHashTable()**, but creates the hash table in *zone* instead of in the default zone. (If *zone* is NULL, the default zone is used.)

NSHashTable ***NSCopyHashTableWithZone**(NSHashTable **table*,
NSZone **zone*)

Returns a pointer to a new copy of *table*, created in *zone* and containing copies of *table*'s pointers to data elements. If *zone* is NULL, the default zone is used.

Free a Table

void **NSFreeHashTable**(NSHashTable **table*)

Releases each element of the specified hash table and frees the table itself.

void **NSResetHashTable**(NSHashTable **table*)

Releases each element but doesn't deallocate the table. This is useful for preserving the table's capacity.

Compare Two Tables

BOOL **NSCompareHashTables**(NSHashTable **table1*, NSHashTable **table2*)

Returns YES if the two hash tables are equal—that is, if each element of *table1* is in *table2*, and the two tables are the same size.

Get the Number of Items

unsigned **NSCountHashTable**(NSHashTable **table*)

Returns the number of elements in *table*.

Retrieve Items

void ***NSHashGet**(NSHashTable **table*, const void **pointer*)

Returns the pointer in the table that matches *pointer* (as defined by the **isEqual()** call-back function). If there is no matching element, the function returns NULL

NSArray ***NSAllHashTableObjects**(NSHashTable **table*)

Returns an array object containing all the elements of *table*. This function should be called only when the table elements are objects, not when they're any other data type.

NSHashEnumerator **NSEnumerateHashTable**(NSHashTable **table*)

Returns an NSHashEnumerator structure that will cause successive elements of *table* to be returned each time this enumerator is passed to **NSNextHashEnumeratorItem()**.

void ***NSNextHashEnumeratorItem**(NSHashEnumerator **enumerator*)

Returns the next element in the table that *enumerator* is associated with, or NULL if *enumerator* has already iterated over all the elements.

Add or Remove an Item

void **NSHashInsert**(NSHashTable **table*, const void **pointer*)

Inserts *pointer*, which must not be NULL, into *table*. If *pointer* matches an item already in the table, the previous pointer is released using the **release()** call-back function that was specified when the table was created.

void NSHashInsertKnownAbsent(NSHashTable *_table_, const void **pointer*)

Inserts *pointer*, which must not be NULL, into *table*. Unike **NSHashInsert()**, this function raises NSInvalidArgumentException if *table* already includes an element that matches *pointer.*

void *NSHashInsertIfAbsent(NSHashTable *table, const void *pointer)

If *pointer* matches an item already in *table*, this function returns the pre-existing pointer; otherwise, it adds *pointer* to the table and returns NULL.

void **NSHashRemove**(NSHashTable **table*, const void **pointer*)

If *pointer* matches an item already in *table*, this function releases the pre-existing item.

Get a String Representation

NSString ***NSStringFromHashTable**(NSHashTable **table*)

Returns a string describing the hash table's contents. The function iterates over the table's elements, and for each one appends the string returned by the **describe()** call-back function. If NULL was specified for the call-back function, the hexadecimal value of each pointer is added to the string.

Map Table Functions

Create a Table

NSMapTable ***NSCreateMapTable**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned *capacity*)

Creates, and returns a pointer to, an NSMapTable in the default zone; the table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types and Constants") that are very similar to the call-back structure used by **NSCreateHashTable()**; in fact, they have the same defaults as documented for that function.

NSMapTable ***NSCreateMapTableWithZone**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned *capacity*, NSZone **zone*)

Like **NSCreateMapTable()**, but creates the map table in *zone* instead of in the default zone. (If *zone* is NULL, the default zone is used.)

NSMapTable ***NSCopyMapTableWithZone**(NSMapTable **table*, NSZone **zone*)

Returns a pointer to a new copy of *table*, created in *zone* and containing copies of *table*'s key and value pointers. If *zone* is NULL, the default zone is used.

Free a Table

void **NSFreeMapTable**(NSMapTable **table*)
Releases each key and value of the specified map table and frees the table itself.

void **NSResetMapTable**(NSMapTable **table*)
Releases each key and value but doesn't deallocate the table. This is useful for preserving the table's capacity.

Compare Two Tables:

BOOL **NSCompareMapTables**(NSMapTable **table1*,
NSMapTable **table2*)
Returns YES if each key of *table1* is in *table2*, and the two tables are the same size. Note that this function does *not* compare values, only keys.

Get the Number of Items

unsigned **NSCountMapTable**(NSMapTable **table*)
Returns the number of key/value pairs in *table*.

Retrieve Items

BOOL **NSMapMember**(NSMapTable **table*,
const void **key*,
void ***originalKey*,
void ***value*)
Returns YES if *table* contains a key equal to *key*. If so, *originalKey* is set to *key*, and *value* is set to the value that the table maps to *key*.

void ***NSMapGet**(NSMapTable **table*,
const void **key*)
Returns the value that *table* maps to *key*, or NULL if the table doesn't contain *key*.

NSMapEnumerator **NSEnumerateMapTable**(NSMapTable **table*)
Returns an NSMapEnumerator structure that will cause successive key/value pairs of *table* to be visited each time this enumerator is passed to **NSNextMapEnumeratorPair()**.

BOOL **NSNextMapEnumeratorPair**(NSMapEnumerator **enumerator*,
void ***key*,
void ***value*)
Returns NO if *enumerator* has already iterated over all the elements in the table that *enumerator* is associated with. Otherwise, this function sets *key* and *value* to match the next key/value pair in the table, and returns YES.

NSArray ***NSAllMapTableKeys**(NSMapTable **table*)
Returns an array object containing all the keys in *table*. This function should be called only when the table keys are objects, not when they're any other type of pointer.

NSArray ***NSAllMapTableValues**(NSMapTable **table*)

Returns an array object containing all the values in *table*. This function should be called only when the table values are objects, not when they're any other type of pointer.

Add or Remove an Item

void **NSMapInsert**(NSMapTable **table*,
const void **key*,
const void **value*)

Inserts *key* and *value* into *table*. If *key* matches a key already in the table, *value* is retained and the previous value is released, using the retain and release call-back functions that were specified when the table was created. Raises NSInvalidArgumentException if *key* is equal to the notAKeyMarker field of the table's NSMapTableKeyCallBacks structure.

void ***NSMapInsertIfAbsent**(NSMapTable **table*,
const void **key*,
const void **value*)

If *key* matches a key already in *table*, this function returns the pre-existing key; otherwise, it adds *key* and *value* to the table and returns NULL. Raises NSInvalidArgumentException if *key* is equal to the notAKeyMarker field of the table's NSMapTableKeyCallBacks structure.

void **NSMapInsertKnownAbsent**(NSMapTable **table*,
const void **key*,
const void **value*)

Inserts *key* (which must not be notAKeyMarker) and *value* into *table*. Unike **NSMapInsert()**, this function raises NSInvalidArgumentException if *table* already includes a key that matches *key*.

void **NSMapRemove**(NSMapTable **table*,
const void **key*)

If *key* matches a key already in *table*, this function releases the pre-existing key and its corresponding value.

NSString ***NSStringFromMapTable**(NSMapTable **table*)

Returns a string describing the map table's contents. The function iterates over the table's key/value pairs, and for each one appends the string "*a* = *b*;\n", where *a* and *b* are the key and value strings returned by the corresponding **describe()** call-back functions. If NULL was specified for the call-back function, a and b are the key and value pointers, expressed as hexadecimal numbers.

Miscellaneous Functions

Get Information about a User

NSString *__NSUserName__(void)

NSString *__NSHomeDirectory__(void)

NSString *__NSHomeDirectoryForUser__(NSString * *userName*)

Log an Error Message

void **NSLog(**NSString *format, ...**)**

Writes to stderr an error message of the form: "*time processName processID format*". The format argument to **NSLog()** is a format string in the style of the standard C function **printf()**, followed by an arbitrary number of arguments that match conversion specifications (such as **%s** or **%d**) in the format string. (You can pass an object in the list of arguments by specifying **%@** in the format string—this conversion specification gets replaced by the string that the object's **description** method returns.)

void **NSLogv(**NSString **format*, va_list *args***)**

Like **NSLog()**, but the arguments to the format string are passed in a single va_list, in the manner of **vprintf()**.

Get Localized Versions of Strings

NSString ***NSLocalizedString(**NSString **key*, NSString **comment***)**

Returns a localized version of the string designated by *key*. The default string table (**Localizable.strings**) in the main bundle is searched for *key*. *comment* is ignored, but can provide information for translators.

NSString ***NSLocalizedStringFromTable(**NSString **key*, NSString **tableName*, NSString **comment***)**

Like **NSLocalizedString()**, but searches the specified table.

NSString ***NSLocalizedStringFromTableInBundle(**NSString **key*, NSString **tableName*, NSBundle **aBundle*, NSString **comment***)**

Like **NSLocalizedStringFromTable**, but uses the specified bundle instead of the application's main bundle.

Convert to and from a String

Class **NSClassFromString(**NSString **aClassName***)** Returns the class object named by *aClassName*, or nil if none by this name is currently loaded.

SEL **NSSelectorFromString(**NSString **aSelectorName***)**
Returns the selector named by *aSelectorName*, or zero if none by this name exists.

NSString ***NSStringFromClass(**Class *aClass***)** Returns an NSString containing the name of *aClass*.

NSString ***NSStringFromSelector(**SEL *aSelector***)** Returns an NSString containing the name of *aSelector.*

Compose a Message To Be Sent Later to an Object

NSInvocation ***NS_INVOCATION(**Class *aClass*, *instanceMessage***)** Returns an NSInvocation object which you can later ask to dispatch *instanceMessage* to an instance of *aClass.* (You later use NSInvocation's **setTarget:** method to make a specific instance of *aClass* the receiver of the message, after which you use **invoke** to cause the message to be sent and **getReturnValue:** to retrieve the result.) Because this is a macro, *message* can be any Objective C message understood by an instance of *aClass*, even a message with multiple arguments.

NSInvocation ***NS_MESSAGE(**id *anObject*, *instanceMessage***)** Like **NS_INVOCATION()**, but the first argument is an instance of a class, rather than a class. The target of the message will be *anObject,* so later you don't use **setTarget:**, only **invoke** and **getReturnValue:**.

Types and Constants

Exception Handling

typedef struct _NSHandler **NSHandler**;	Exception handler information.
typedef volatile void **NSUncaughtExceptionHandler**(NSException **exception*);	Register an uncaught exception handler.
NSString ***NSInconsistentArchiveException**;	Consistency error in archive file.
NSString ***NSGenericException**;	General programming error.
NSString ***NSInternalInconsistencyException**;	Some item that should be invariant changed.
NSString ***NSInvalidArgumentException**;	Invalid argument.
NSString ***NSMallocException**;	No memory left to allocate.
NSString ***NSRangeException**;	Attempt to access an element beyond the limit of an array or similar structure.
NSString ***NSByteStoreLockedException**;	
NSString ***NSByteStoreVersionException**;	
NSString ***NSBTreeStoreKeyTooLargeException**;	
NSString ***NSByteStoreDamagedException**;	

Geometry

```
typedef struct _NSPoint {
        float x;
        float y;
} NSPoint;
```

Point definition.

```
typedef struct _NSSize {
        float width;
        float height;
} NSSize;
```

Rectangle sizes.

typedef struct _NSRect {
    NSPoint **origin**;
    NSSize **size**;
} **NSRect**;
	Rectangle.

typedef enum _NSRectEdge {
    NSMinXEdge,
    NSMinYEdge,
    NSMaxXEdge,
    NSMaxYEdge
} **NSRectEdge**;
	Sides of a rectangle.

const NSPoint **NSZeroPoint**;	A zero point.
const NSRect **NSZeroRect**;	A zero origin rectangle.
const NSSize **NSZeroSize**;	A zero size rectangle.

Hash Table

typedef struct **NSHashEnumerator**;	Private type for enumerating.
typedef struct _NSHashTable **NSHashTable**;	Hash table type.
typedef struct {	Callback functions.
unsigned (***hash**)(NSHashTable **table*, const void **anObject*);	Hashing function. Note: Elements with equal values must have equal hash function values.
BOOL (***isEqual**)(NSHashTable **table*, const void **anObject*, const void **anObject*);	Comparison function.
void (***retain**)(NSHashTable **table*, const void **anObject*);	Retaining function called when adding elements to table.
void (***release**)(NSHashTable **table*, void **anObject*);	Releasing function called when a data element is removed from the table.
NSString *(***describe**)(NSHashTable **table*, const void **anObject*);	Description function.
} **NSHashTableCallBacks**;	

const NSHashTableCallBacks **NSIntHashCallBacks**; For sets of pointer-sized or smaller quantities.

const NSHashTableCallBacks **NSNonOwnedPointerHashCallBacks**;
For sets of pointers hashed by address.

const NSHashTableCallBacks **NSNonRetainedObjectHashCallBacks**;
For sets of objects without retaining and releasing.

const NSHashTableCallBacks **NSObjectHashCallBacks**;
For sets of objects; similar to NSSet.

const NSHashTableCallBacks **NSOwnedPointerHashCallBacks**;
For sets of pointers with transfer of ownership upon insertion.

const NSHashTableCallBacks **NSPointerToStructHashCallBacks**;
For sets of pointers to structs when the first field of the struct is the size of an **int**.

Map Table

typedef struct **NSMapEnumerator**; Private type for enumerating.

typedef struct _NSMapTable **NSMapTable**; Map table type.

typedef struct { Callback functions for a key.
unsigned (***hash**)(NSMapTable **table*, const void **anObject*);
Hashing function. Note: Elements with equal values must have equal hash function values.
BOOL (***isEqual**)(NSMapTable **table*, const void **anObject*, const void **anObject*);
Comparison function.
void (***retain**)(NSMapTable **table*, const void **anObject*);
Retaining function called when adding elements to table.
void (***release**)(NSMapTable **table*, void **anObject*);
Releasing function called when a data element is removed from the table.
NSString *(***describe**)(NSMapTable **table*, const void **anObject*);
Description function.
const void ***notAKeyMarker**; Quantity that is not a key to the hash table.
} **NSMapTableKeyCallBacks**;

typedef struct { — Callback functions for a value.

    void (*__retain__)(NSMapTable **table*, const void **anObject*);
    — Retaining function called when adding elements to table.

    void (*__release__)(NSMapTable **table*, void **anObject*);
    — Releasing function called when a data element is removed from the table.

    NSString *(*__describe__)(NSMapTable **table*, const void **anObject*);
    — Description function.

} **NSMapTableValueCallBacks**;

#define **NSNotAnIntMapKey**; — Quantity that is never a map key.

#define **NSNotAPointerMapKey**; — Quantity that is never a map key.

const NSMapTableKeyCallBacks **NSIntMapKeyCallBacks**;
For keys that are pointer-sized or smaller quantities.

const NSMapTableValueCallBacks **NSIntMapValueCallBacks**;
For values that are pointer-sized quantities.

const NSMapTableKeyCallBacks **NSNonOwnedPointerMapKeyCallBacks**;
For keys that are pointers not freed.

const NSMapTableValueCallBacks **NSNonOwnedPointerMapValueCallBacks**;
For values that are owned pointers.

const NSMapTableKeyCallBacks **NSNonOwnedPointerOrNullMapKeyCallBacks**;
For keys that are pointers not freed, or NULL.

const NSMapTableKeyCallBacks **NSNonRetainedObjectMapKeyCallBacks**;
For sets of objects without retaining and releasing.

const NSMapTableKeyCallBacks **NSObjectMapKeyCallBacks**;
For keys that are objects.

const NSMapTableValueCallBacks **NSObjectMapValueCallBacks**;
For values that are objects.

const NSMapTableKeyCallBacks **NSOwnedPointerMapKeyCallBacks**;
For keys that are pointers with transfer of ownership upon insertion.

const NSMapTableValueCallBacks **NSOwnedPointerMapValueCallBacks**;
For values that are owned pointers.

Notification Queue

```
typedef enum {
        NSPostWhenIdle,
        NSPostASAP,
        NSPostNow
} NSPostingStyle;
```

NSPostWhenIdle	Post the notification when the run loop is idle.
NSPostASAP	Post the notification as soon as possible.
NSPostNow	Post the notification immediately.

```
typedef enum {
        NSNotificationNoCoalescing,
        NSNotificationCoalescingOnName,
        NSNotificationCoalescingOnSender,
} NSNotificationCoalescing;
```

NSNotificationNoCoalescing	Do not coalesce similar notifications in the queue.
NSNotificationCoalescingOnName	Coalesce notifications in the queue matching name.
NSNotificationCoalescingOnSender	Coalesce notifications in the queue matching sender.

Run Loop

NSString *__NSConnectionReplyMode__;	NSRunLoop mode in which Distributed Object system seeks replies.
NSString *__NSDefaultRunLoopMode__;	Common NSRunLoop mode.

Search Results

```
typedef enum _NSComparisonResult {
      NSOrderedAscending,
      NSOrderedSame,
      NSOrderedDescending
} NSComparisonResult;
```

Ordered comparison results.

```
enum {
      NSCaseInsensitiveSearch,
      NSLiteralSearch,
      NSBackwardsSearch,
      NSAnchoredSearch

};
```

Flags passed to various search methods.

enum {**NSNotFound**};	Indicates an item not found.

String

typedef unsigned **NSStringEncoding**;	Known encodings.

enum
    NSASCIIStringEncoding,
    NSNEXTSTEPStringEncoding,
    NSJapaneseEUCStringEncoding,
    NSUTF8StringEncoding,
    NSISOLatin1StringEncoding ,
    NSSymbolStringEncoding ,
    NSNonLossyASCIIStringEncoding,
    NSShiftJISStringEncoding,
    NSISOLatin2StringEncoding,
    NSUnicodeStringEncoding
};
	Known encodings.

enum _NSOpenStepUnicodeReservedBase {
    NSOpenStepUnicodeReservedBase
};
	Base for Unicode characters.

NSHashStringLength;	Hash string length.
NSMaximumStringLength;	Maximum string length.

Threads

typedef enum {
    NSInteractiveThreadPriority,
    NSBackgroundThreadPriority,
    NSLowThreadPriority
} **NSThreadPriority**;
	Thread priorities.

NSString ***NSBecomingMultiThreaded**;	Notifications.
NSString ***NSThreadExiting**;	

User Defaults

NSString ***NSArgumentDomain**;	For defaults parsed from the application's arguments.
NSString ***NSGlobalDomain**;	For defaults seen by all applications.
NSString ***NSRegistrationDomain**;	For registered defaults.
NSString ***NSUserDefaultsChanged**;	Public notification.

NSString ***NSWeekDayNameArray**;	Keys for language-dependent information.
NSString ***NSShortWeekDayNameArray**;	
NSString ***NSMonthNameArray**;	
NSString ***NSShortMonthNameArray**;	
NSString ***NSTimeFormatString**;	
NSString ***NSDateFormatString**;	
NSString ***NSTimeDateFormatString**;	
NSString ***NSShortTimeDateFormatString**;	
NSString ***NSCurrencySymbol**;	
NSString ***NSDecimalSeparator**;	
NSString ***NSThousandsSeparator**;	
NSString ***NSInternationalCurrencyString**;	
NSString ***NSCurrencyString**;	
NSString ***NSDecimalDigits**;	
NSString ***NSAMPMDesignation**;	

Miscellaneous

typedef struct {
    int offset;
    int size;
    char *type;
} **NSArgumentInfo**;
	Specifies layout of arguments used in invocations.

typedef struct _NSRange {
    unsigned int **location**;
    unsigned int **length**;
} **NSRange**;
	Specifies a range of items in arrays, strings, and so on.

typedef double **NSTimeInterval**;	Time interval difference between two dates.
typedef struct _NSZone **NSZone**;	Large region allocation.

typedef int **NSBTreeComparator**(NSData *, NSData *, const void *);

3 *Display PostScript*

Classes

Classes listed here and the protocol in the following section constitute OpenStep's object-oriented interface to the Display PostScript System. As such, many of the argument and return types that appear below (specifically, those having a "DPS" prefix) are not described in this document. Rather, they are detailed in the specification for the Display PostScript System itself, as found in the *Display PostScript System, Client Library Reference Manual*, by Adobe Systems Incorporated.

NSDPSContext

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: DPSClient/NSDPSContext.h

Class Description

The NSDPSContext class is the programmatic interface to objects that represent Display PostScript System *contexts*. A context can be thought of as a *destination* to which PostScript code is sent for execution. Each Display PostScript context contains its own complete PostScript environment including its own local VM (PostScript Virtual Memory). Every context has its own set of stacks, including an operand stack, graphics state stack, dictionary stack, and execution stack. Every context also contains a **FontDirectory** which is local to that context, plus a **SharedFontDirectory** that is shared across all contexts. There are three built-in dictionaries in the dictionary stack. From top to bottom, they are **userdict**, **globaldict**, and **systemdict**. **userdict** is private to the context, while

globaldict and **systemdict** are shared by all contexts. **globaldict** is a modifiable dictionary containing information common to all contexts. **systemdict** is a read-only dictionary containing all the PostScript operators.

At any time there is the notion of the *current context*. The current context for the current thread may be set using **setCurrentContext:.**

NSDPSContext objects by default write their output to a specified *data* destination. This is used for printing, FAXing, and for generation of saved EPS (Encapsulated PostScript) code. The means to create contexts that interact with displays are platform-specific.

The NSApplication object creates a context by default.

NSDPSContext Objects and Display PostScript System Context Records

When an NSDPSContext object is created, it creates and manages a *DPSContext* record. Programmers familiar with the client side C function interface to the Display PostScript System can access the DPSContext record by sending a **context** message to an NSDPSContext object. You can then operate on this context record using any of the functions or single operator functions defined in the Display PostScript System client library. Conversely, you can create an NSDPSContext object from a DPSContext record with the **DPSContextObject()** function, as defined in "Client Library Functions". You can then work with the created NSDPSContext object using any of the methods described here.

General Exception Conditions

A variety of exceptions can be raised from NSDPSContext. In most cases, exceptions are raised because of errors returned from the Display PostScript Server. Exceptions are listed under "Types and Constants." Also see the *Display PostScript System, Client Library Reference Manual*, by Adobe Systems Incorporated, for more details on Display PostScript System error names and their possible causes.

Initializing a Context

– **initWithMutableData:**(NSMutableData *)*data*
    forDebugging:(BOOL)*debug*
    languageEncoding:(DPSProgramEncoding)*langEnc*
    nameEcoding:(DPSNameEncoding)*nameEnc*
    textProc:(DPSTextProc)*tProc*
    errorProc:(DPSErrorProc)*errorProc*

Initializes a newly allocated NSDPSContext that writes its output to *data* using the language and name encodings specified by *langEnc* and *nameEnc*. The callback functions *tProc* and *errorProc* handle text and errors generated by the context. If *debug* is YES, the output is given in human-readable form in which large structures (such as images) may be represented by comments.

Testing the Drawing Destination

– (BOOL)**isDrawingToScreen**

Returns YES if the drawing destination is the screen.

Accessing Context Data

– (NSMutableData *)**mutableData** — Returns the receiver's data object.

Setting and Identifying the Current Context

+ (NSDPSContext *)**currentContext** — Returns the current context of the current thread.

+ (void)**setCurrentContext:**(NSDPSContext *)*context*
  Installs *context* as the current context of the current thread.

– (DPSContext)**DPSContext** — Returns the corresponding DPScontext.

Controlling the Context

– (void)**flush** — Forces any buffered data to be sent to its destination.

– (void)**interruptExecution** — Interrupts execution in the receiver's context.

– (void)**notifyObjectWhenFinishedExecuting:**(id <NSDPSContextNotification>)*object*
  Registers *object* to receive a **contextFinishedExecuting**: message when the NSDPSContext's destination is ready to receive more input.

– (void)**resetCommunication** — Discards any data that hasn't already been sent to its destination.

– (void)**wait** — Waits until the NSDPSContext's destination is ready to receive more input.

Managing Returned Text and Errors

+ (NSString *)**stringForDPSError:**(const DPSBinObjSeqRec *)*error*
  Returns a string representation of *error.*

– (DPSErrorProc)**errorProc** — Returns the context's error callback function.

– (void)**setErrorProc:**(DPSErrorProc)*proc* — Sets the context's error callback function to *proc*.

– (void)**setTextProc:**(DPSTextProc)*proc* — Sets the context's text callback function to *proc*.

– (DPSTextProc)**textProc** — Returns the context's text callback function.

Sending Raw Data

– (void)**printFormat:**(NSString *)*format,...* — Constructs a string from *format* and following string objects (in the manner of **printf()**) and sends it to the context's destination.

– (void)**printFormat:**(NSString *)*format* **arguments:**(va_list)*argList*

Constructs a string from *format* and *argList* (in the manner of **vprintf()**) and sends it to the context's destination.

– (void)**writeData:**(NSData *)*buf*

Sends the PostScript data in *buf* to the context's destination.

– (void)**writePostScriptWithLanguageEncodingConversion:**(NSData *)*buf*

Writes the PostScript data in *buf* to the context's destination. The data, formatted as plain text, encoded tokens, or a binary object sequence, is converted as necessary depending on the language encoding of the receiving context.

Managing Binary Object Sequences

– (void)**awaitReturnValues**

Waits for all return values from the result table.

– (void)**writeBOSArray:**(const void *)*data* **count:**(unsigned int)*items* **ofType:**(DPSDefinedType)*type*

Write an array to the context's destination as part of a binary object sequence. The array is taken from *data* and consists of *items* items of type *type*.

– (void)**writeBOSNumString:**(const void *)*data* **length:**(unsigned int)*count* **ofType:**(DPSDefinedType)*type* **scale:**(int)*scale*

Write a number string to the context's destination as part of a binary object sequence. The string is taken from *data* as described by *count*, *type,* and *scale.*

– (void)**writeBOSString:**(const void *)*data* **length:**(unsigned int)*bytes*

Write a string to the context's destination as part of a binary object sequence. The string is taken from *bytes* (a count) of *data.*

– (void)**writeBinaryObjectSequence:**(const void *)*data* **length:**(unsigned int)*bytes*

Write a binary object sequence to the context's destination. The sequence consists of *bytes* (a count) of *data.*

– (void)**updateNameMap**

Updates the context's name map from the client library's name map.

Managing Chained Contexts

– (void)**chainChildContext:**(NSDPSContext *)*child*

Links *child* (and all of it's children) to the receiver as its chained context, a context that receives a copy of all PostScript code sent to the receiver.

– (NSDPSContext *)**childContext**

Returns the receiver's child context, or **nil** if none exists.

– (NSDPSContext *)**parentContext**

Returns the receiver's parent context, or **nil** if none exists.

– (void)**unchainContext**

Unlinks the child context (and all of it's children) from the receiver's list of chained contexts.

Debugging Aids

+ (BOOL)**areAllContextsOutputTraced**
Returns YES if the data flowing between the application's contexts and their destinations is copied to diagnostic output.

+ (BOOL)**areAllContextsSynchronized**
Returns YES if all NSPDSContext objects invoke the **wait** method after sending each batch of output.

+ (void)**setAllContextsOutputTraced:**(BOOL)*flag*
Causes the data (PostScript code, return values, etc.) flowing between the all the application's contexts and their destinations to be copied to diagnostic output.

+ (void)**setAllContextsSynchronized:**(BOOL)*flag*
Causes the **wait** method to be invoked each time an NSDPSContext object sends a batch of output to its destination.

– (BOOL)**isOutputTraced**
Returns YES if the data flowing between the application's single context and its destination is copied to diagnostic output.

– (BOOL)**isSynchronized**
Returns whether the **wait** method is invoked each time the receiver sends a batch of output to the server.

– (void)**setOutputTraced:**(BOOL)*flag*
Causes the data (PostScript code, return values, etc.) flowing between the application's single context and the Display PostScript server to be copied to diagnostic output.

– (void)**setSynchronized:**(BOOL)*flag*
Sets whether the **wait** method is invoked each time the receiver sends a batch of output to its destination.

Protocols

NSDPSContextNotification

Adopted By: no OpenStep classes

Declared In: DPSClient/NSDPSContext.h

Protocol Description

The NSDPSContextNotification protocol supplies information about the execution status of a sequence of PostScript commands previously sent to the Display PostScript server.

Synchronizing Application and Display PostScript Server Execution

– (void)**contextFinishedExecuting:**(NSDPSContext *)*context*

Notifies the receiver that the context has finished executing a batch of PostScript commands. See **notifyObjectWhenFinishedExecuting:** (NSDPSContext).

Display PostScript Operators

The *PostScript Language Reference Manual*, *Second Edition*, by Adobe Systems Incorporated, provides the specifications for standard PostScript and Display PostScript operators. Listed here are operators found in OpenStep but not in the standard implementation of the PostScript language.

Compositing Operators

src_x src_y width height srcgstate $dest_x$ $dest_y$ op	**composite**	–	Composites rectangle in source graphics state with image in current window.
$dest_x$ $dest_y$ width height op	**compositerect**	–	Composites rectangle of current color and coverage with image in current graphics state.
src_x src_y width height srcgstate $dest_x$ $dest_y$ delta	**dissolve**	–	Dissolves between area of window referred to by *srcgstate* and equal area of window referred to by the current graphics state.

Graphics State Operators

coverage	**setalpha**	–	Sets the current coverage.
–	**currentalpha**	*coverage*	Returns the current coverage setting.

Client Library Functions

The Display PostScript Client Library is composed of system-dependent and a system-independent parts. The *Display PostScript System, Client Library Reference Manual*, by Adobe Systems, Incorporated., provides the specification for the system-independent portion of this library.

Functions that are part of OpenStep's system-dependent part of the Display PostScript Client Library are listed here.

PostScript Execution Context Functions

Convert a DPSContext to an NSDPSContext Object

NSDPSContext *__DPSContextObject(__DPSContext *ctxt*__)__

Communication with the Display PostScript Server

Send a PostScript User Path to the Display PostScript Server

These functions are used to send a user path, plus one other *action*, to the Display PostScript Server. In the **…WithMatrix** forms of these operators, the *matrix* operand is the optional matrix argument used by the **ustroke**, **inustroke**, and **ustrokepath** operators. The *matrix* argument may be NULL, in which case it is ignored.

void **PSDoUserPath**(const void **coords*, int *numCoords*, DPSNumberFormat *numType*, const DPSUserPathOp **ops*, int *numOps*, const void **bbox*, DPSUserPathAction *action*)

void **PSDoUserPathWithMatrix(**void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, DPSUserPathAction *action*, float *matrix*[6]**)**

void **DPSDoUserPath**(DPSContext *context*, const void **coords*, int *numCoords*, DPSNumberFormat *numType*, const DPSUserPathOp **ops*, int *numOps*, const void **bbox*, DPSUserPathAction *action*) void **DPSDoUserPathWithMatrix(**DPSContext *context***,** void **coords*, int *numCoords*, DPSNumberFormat *numType*, unsigned char **ops*, int *numOps*, void **bbox*, DPSUserPathAction *action*, float *matrix*[6]**)**

Send PostScript Code to the Display PostScript Server

void **PSFlush(**void**)**
void **PSWait(**void**)**

Single-Operator Functions

Single-operator functions provide a C language interface to the individual operators of the PostScript language. The specification for a single-operator function is identical to that of the PostScript operator it represents. The *PostScript Language Reference Manual*, *Second Edition*, by Adobe Systems Incorporated, provides the specifications of all standard PostScript operators. Also refer to the *Display PostScript System*, *Client Library Reference Manual*, by Adobe Systems Incorporated. Listed below are single-operator functions that correspond to operators found in OpenStep but not in the standard implementation of the PostScript language.

These functions have either a "PS" or a "DPS" prefix. For every single-operator function with a "PS" prefix, there's a corresponding single-operator function with a "DPS" prefix. The PS and DPS functions are identical except that DPS functions take an additional (first) argument that represents the PostScript execution context.

Besides using standard C language types, some single-operator functions use **userobject—**an **int** that refers to the value returned by **DPSDefineUserObject()**.

In the function descriptions below, *x* and *y* refer to the origin of *source* rectangles, and *w* and *h* refer to the width and height of the source rectangles. *gstateNum* refers to the graphics state (gstate) of the source rectangle. *dx* and *dy* refer to the origin of the *destination* for the compositing or dissolving operation. *op* refers to the specific compositing operation. *a* or *alpha* refers to the coverage component used for compositing operations.

"PS" Prefix Functions

void **PScomposite(**float *x*, float *y*, float *w*, float *h*, int *gstateNum*, float *dx*, float *dy*, int *op***)**

void **PScompositerect(**float *x*, float *y*, float *w*, float *h*, int *op***)**

void **PScurrentalpha(**float **alpha***)**

void **PSdissolve(**float *x*, float *y*, float *w*, float *h*, int *gstateNum*, float *dx*, float *dy*, float *delta***)**

void **PSsetalpha(**float *a***)**

"DPS" Prefix Functions

void **DPScomposite(**DPSContext *ctxt*, float *x*, float *y*, float *w*, float *h*, int *gstateNum*, float *dx*, float *dy*, int *op***)**

void **DPScompositerect(**DPSContext *ctxt*, float *dx*, float *dy*, float *w*, float *h*, int *op***)**

void **DPScurrentalpha(**DPSContext *ctxt*, float **pcoverage***)**

void **DPSdissolve(**DPSContext *ctxt*, float *x*, float *y*, float *w*, float *h*, int *gstateNum*, float *dx*, float *dy*, float *delta***)**

void **DPSsetalpha(**DPSContext *ctxt*, float *a***)**

Types and Constants

The Display PostScript Client Library is composed of system-dependent and a system-independent parts. The *Display PostScript System, Client Library Reference Manual*, by Adobe Systems, Incorporated, provides the specification for the system-independent portion of this library.

The defined types, enumeration constants, and global variables that are part of OpenStep's system-dependent part of the Display PostScript Client Library are listed here.

Defined Types

Number Formats

typedef enum _DPSNumberFormat {

#ifdef __BIG_ENDIAN__
    dps_float = 48,
    dps_long = 0,
    dps_short = 32
#else
    dps_float = 48+128,
    dps_long = 0+128,
    dps_short = 32+128
#endif
} **DPSNumberFormat**;

Other permitted values are:

- For 32-bit fixed-point numbers, use **dps_long** plus the number of bits in the fractional part.
- For 16-bit fixed-point numbers, use **dps_short** plus the number of bits in the fractional part.

Backing Store Types

typedef enum _NSBackingStoreType {
        NSBackingStoreRetained,
        NSBackingStoreNonretained,
        NSBackingStoreBuffered
} **NSBackingStoreType**;

Compositing Operations

```
typedef enum _NSCompositingOperation {
    NSCompositeClear,
    NSCompositeCopy,
    NSCompositeSourceOver,
    NSCompositeSourceIn,
    NSCompositeSourceOut,
    NSCompositeSourceAtop,
    NSCompositeDataOver,
    NSCompositeDataIn,
    NSCompositeDataOut,
    NSCompositeDataAtop,
    NSCompositeXOR,
    NSCompositePlusDarker,
    NSCompositeHighlight,
    NSCompositePlusLighter
} NSCompositingOperation;
```

Window Ordering

```
typedef enum _NSWindowOrderingMode {
    NSWindowAbove,
    NSWindowBelow,
    NSWindowOut
} NSWindowOrderingMode;
```

User Path Operators

These constants define the operator numbers used to construct the operator array parameter of DPSDoUserPath.

```
typedef unsigned char DPSUserPathOp;
enum {
    dps_setbbox,
    dps_moveto,
    dps_rmoveto,
    dps_lineto,
    dps_rlineto,
    dps_curveto,
    dps_rcurveto,
    dps_arc,
    dps_arcn,
    dps_arct,
    dps_closepath,
```

dps_ucache
};

User Path Actions

These constants define the action of a DPSDoUserPath. In addition to the actions defined here, any other system name index may be used. See the *PostScript Language Reference Manual*, *Second Edition*, by Adobe Systems Incorporated, for a detailed list of system name indexes.

typedef enum _DPSUserPathAction {
dps_uappend,
dps_ufill,
dps_ueofill,
dps_ustroke,
dps_ustrokepath,
dps_inufill,
dps_inueofill,
dps_inustroke,
dps_def,
dps_put
} **DPSUserPathAction**;

Enumerations

Special Values for Alpha

enum {
NSAlphaEqualToData,
NSAlphaAlwaysOne
};

User Object Representing the PostScript Null Object

enum {
DPSNullObject
};

Symbolic Constants

Error Code Base

DPS_OPENSTEP_ERROR_BASE

Global Variables

Exception Names

NSString ***DPSPostscriptErrorException;**
NSString ***DPSNameTooLongException;**
NSString ***DPSResultTagCheckException;**
NSString ***DPSResultTypeCheckException;**
NSString ***DPSInvalidContextException;**
NSString ***DPSSelectException;**
NSString ***DPSConnectionClosedException;**
NSString ***DPSReadException;**
NSString ***DPSWriteException;**
NSString ***DPSInvalidFDException;**
NSString ***DPSInvalidTEException;**
NSString ***DPSInvalidPortException;**
NSString ***DPSOutOfMemoryException;**
NSString ***DPSCantConnectException;**