Chapter 3

Converting Application Kit Classes

This chapter describes changes to the Application Kit that affect only the Application Kit. The changes to the Application Kit are many and varied: there are stylistic changes, which are described in the chapter ªGlobal API and Style Changes,º and Application Kit objects now use Foundation objects where they used to use Common objects, which is described in the chapter ªConverting the Common Classes.º Look in this chapter for changes that affect only the Application Kit itself and not its interaction with other frameworks.

Application Conversion

Stage 5

There are two big changes at the workspace and application levels:

SquareBullet.eps ¬The workspace is now represented by an NSWorkspace object, which replaces the NXWorkspaceRequestProtocol protocol. You access the workspace with the message **[NSWorkspace sharedWorkspace]**.

SquareBullet.eps ¬The NSApplication object is decentralized. Many Application methods are obsolete and replaced by methods in other classes. The table below provides the details of this change, although the conversion process makes most of these substitutions for you. The rest of this section describes conversions you must perform yourself.

Action TableHeadRule.eps ¬	Moved to Class	Obsolete Application Methods
Loading nib files TableRule.eps ¬	NSBundle (NSBundleAdditions)	loadNibFile:... loadNibSection:....
Setting global color transparency TableRule.eps ¬	NSColor	setImportAlpha: doesImportAlpha
Finding out host of DPS Server TableRule.eps ¬	NSDPSContext	hostName
Accessing the shared NSPrintInfo object TableRule.eps ¬	NSPrintInfo	printInfo setPrintInfo:
Accessing screens TableRule.eps ¬	NSScreen	mainScreen colorScreen getScreens:count: getScreenSize:
Retrieving user languages TableRule.eps ¬		NSUserDefaults systemLanguages
Finding the focus view TableRule.eps ¬	NSView	focusView
Controlling automatic display updating TableRule.eps ¬	NSWindow	autoUpdate setAutoUpdate:
Accessing the workspace TableRule.eps ¬	NSWorkspace	workspace
Mounting and unmounting disks TableRule.eps ¬	NSWorkspace	mounted: unmounted:

Performing file operations	NSWorkspace	openFile:ok: openTempFile:ok: fileOperationCompleted:

TableRule.eps ¬

Loading nib Files

As shown in the table above, NSBundle now loads nib files. To do so, it uses the following methods:

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

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

Notice that these two methods return **BOOL**. You should inspect places where your application loads nib files and make sure that the return type is tested appropriately.

Setting the Application Icon

Previously, Application's **appIcon** method returned a Window object, allowing you to treat the application icon as a window. Now, the NSApplication object only allows you to set the image on the icon. Use the replacement for **appIcon** shown in the following table.

Obsolete Application Method	**Possible Replacement**
TableHeadRule.eps ¬	
appIcon	setApplicationIconImage:

TableRule.eps ¬

Application Name

NSApplication objects cannot give you their application name. Instead of asking for the application name, you ask for the name of the process from a new Foundation Framework object, NSProcessInfo. NSProcessInfo objects store information about a process. The methods in the following table are obsolete.

Obsolete Application Method TableHeadRule.eps ¬	Possible Replacement
appName TableRule.eps ¬	[[NSProcessInfo processInfo] processName]
setAppName: TableRule.eps ¬	[[NSProcessInfo processInfo] setProcessName:]

Speakers, Listeners, and Journalers

The classes Speaker, Listener, and NXJournaler are obsolete. The Speaker and Listener implementations are still provided for compatibility, but they will not be available in the next release, and they are not in the OpenStep specification. As a result of this change, the Application methods shown in the following table are obsolete.

Obsolete Application Method TableHeadRule.eps ¬	Possible Replacement
isJournalable masterJournaler slaveJournaler	None.

setJournalable: TableRule.eps ¬	
appListener	Use DO.
appListenerPortName	
setAppListener: TableRule.eps ¬	
appSpeaker	Use DO.
replyPort:	
setAppSpeaker: TableRule.eps ¬	

If you are doing a shallow conversion and you want to keep your Speaker and Listener, use the **-n** option to **msgwrap** to generate the Speaker and Listener subclasses. The **-n** option generates code that compiles in this release.

Aborting Processes

Users can no longer use Command-period to abort a process, making the functions in the following table obsolete.

Obsolete Function TableHeadRule.eps ¬	**Possible Replacement**
NXUserAborted() TableRule.eps ¬	None
NXResetUserAbort TableRule.eps ¬	None

Naming Objects

Because of global namespace issues, you can no longer name objects in Interface Builder. The following table lists functions that are obsolete because of this change.

Obsolete Function TableHeadRule.eps ¬	Possible Replacement
NXGetNamedObject() TableRule.eps ¬	Use outlets in Interface Builder.
NXGetObjectName() TableRule.eps ¬	Use outlets in Interface Builder.
NXNameObject() TableRule.eps ¬	Use outlets in Interface Builder.
NXUnnameObject() TableRule.eps ¬	Use outlets in Interface Builder.

Application-Level Events

The NSApplication methods **applicationDefined:** and **powerOff:** are obsolete. OpenStep has no application-defined event type or power-off event type. For power-off events, you can register to receive the NSWorkspaceWillPowerOffNotification.

Miscellaneous Obsolete Methods

The methods listed in the table below are now obsolete because they aren't portable to other window systems.

Obsolete Application Method	Possible Replacement

TableHeadRule.eps ¬	
activeApp TableRule.eps ¬	None.
activate: TableRule.eps ¬	None.
getWindowNumbers:count: TableRule.eps ¬	[NSApp windows] returns an NSArray of NSWindows. If you are not using the strict OpenStep definition, use the functions NSCountWindow() and NSWindowList().

Application Gotchas

After the Application conversion, look out for the following:

Importing Application Kit Headers

If you import individual Application Kit header files rather than importing **AppKit.h**, you may receive compiler errors after this conversion because many of the Application methods have been moved to other classes. If you didn't already import the appropriate header file, the conversion process cannot do it for you. If necessary, consult the table shown at the beginning of this section to find out which header to import. You may want to consider importing **AppKit.h** instead of individual header files to reduce your compile time because **AppKit.h** is precompiled.

Animating the Application Icon

Previously, to animate your application's icon, you drew inside the Window that represented the icon. This Window no longer exists in OpenStep. To animate the icon, you repeatedly send **setApplicationIconImage:**. As a result, animation is much slower, and on some platforms, might not even be possible. You will probably want to limit the amount of animation your icon performs.

Browser Conversion

Stage 5

NXBrowser had three different types of delegates: normal, lazy, and very lazy. Normal delegates created and loaded cells into the browser an entire column at a time. Lazy delegates created cells a column at a time but loaded them only as the browser requested them. Very lazy delegates only told the browser how many cells there were to be in the column and loaded them as the browser requested them.

The new NSBrowser eliminates the lazy class of delegate, allowing only normal and very lazy delegates. Both delegates should implement **browser:willDisplayCell:atRow:column:**. A normal delegate implements **browser:createRowsForColumn:inMatrix:**. (This method now returns **void**.) A very lazy browser implements **browser:numberOfRowsInColumn:**. Browser delegates no longer have to send **setLoaded:** to a cell at any time because the NSBrowser can determine that state by itself. The following table summarizes the changes to browser delegate methods.

Old Method Name	**New Method Name**
TableHeadRule.eps ¬	
browser:fillMatrix:inColumn: TableRule.eps ¬	browser:createRowsForColumn:inMatrix:
browser:loadCell:atRow:inColumn: TableRule.eps ¬	browser:willDisplayCell:atRow:column:

Color Conversion

Stage 3

In OpenStep, colors are represented by NSColor objects rather than NXColor structures. NSColor objects are constant objects that can't be changed; when you modify a color, you create a new NSColor object out of an existing one. However, don't confuse NSColor objects with shared objects. Identical colors are not guaranteed to be the same NSColor instance. This section describes the changes to working with colors where you may need to perform some of the conversion yourself.

Colors as Instance Variables

If you have an NXColor instance variable, you will need to perform some of the conversion yourself, as described here.

1. Change the instance variable from a color structure to an NSColor object. Remember to retain the NSColor object when it is initialized and to release it in the **dealloc** method.

2. If you archived and unarchived the color structure, you will also need to convert that code by hand. The conversion process will change code that unarchives a color structure to this:

   ```
   myColor = [[theUnarchiver decodeNXColor] retain];
   ```

   You may leave the above message alone if you are truly unarchiving an NXColor structure. If you have changed the structure to the NSColor object, though, you should change the message to:

   ```
   myColor = [[theUnarchiver decodeObject] retain];
   ```

Gray Values

Application Kit objects no longer define a separate gray value and color value. You used to be able to specify a color for an object and a separate gray value to be used only on 2-bit grayscale screens. Now, you set only the color of the object. Methods such as **setBackgroundGray:** are obsolete; the conversion process replaces it

with **setBackgroundColor:**. If your application sets both a gray value and a color for an object, you may see two **setBackgroundColor:** messages in a row after this conversion is complete.

The Application Kit used to allow you to set the background gray value to -1.0 to disable the drawing of the background. As a consequence of removing the gray values, Application Kit objects now have the methods **setDrawsBackground:** and **setDrawsCellBackground:** to set whether the background is drawn. The conversion process changes **setBackgroundGray:-1.0** to **setDrawsBackground:NO**. If you stored -1.0 in a variable and passed that variable to **setBackgroundGray:**, the conversion process changes it to **setBackgroundColor:**. Because -1.0 is an illegal color value, this message will be flagged at run time. You will need to change **setBackgroundColor:** to **setDrawsBackground:NO** yourself.

Obsolete Method	**Replacement**
TableHeadRule.eps ¬	
setBackgroundGray: TableRule.eps ¬	setBackgroundColor: or delete if you have already set a separate color
setBackgroundGray:-1.0 TableRule.eps ¬	setDrawsBackground:NO

Alpha Components

All NSColors now have an alpha component. Any instances of **NX_NOALPHA** in your code (now obsolete) are converted to the value 1.0, which indicates complete opaqueness. You can use the NSColor **setIgnoresAlpha:** method to enable or disable the use of the alpha component.

Old Code

```
NXColor *myColor = NXConvertRGBAToColor(a, b, c, NX_NOALPHA);
```

New Code

```
NSColor *myColor;
[NSColor setIgnoresAlpha:YES];
myColor = [NSColor colorWithCalibratedRed:a green:b
     blue:c alpha:1.0]
```

Finding a Color Component

You can only ask for a color component that makes sense in the color's colorspace. For example, if the color is in the RGB colorspace, you can ask only for red, green, or blue components. If you need to ask for a component that is not in the color's colorspace, convert it to another colorspace with the method **colorUsingColorSpaceName:**. This method returns a new NSColor object whose color is the same as the receiver's (if possible) but is in a different colorspace. Having to explicitly convert colors to a new colorspace gives you a chance to find out if the conversion is possible and flags the spots where colors are converted.

The conversion process will change all requests for color components automatically. If it needs to use **colorUsingColorSpaceName:**, it will flag that message. You should make sure that you really do want to convert to a new colorspace.

Changing Color Components

NSColor objects are constant objects that cannot be changed, so you cannot change an NSColor object's color component. If you use one of the functions that changes a color component, instead create a separate NSColor object using one of the several NSColor creation methods. For example, suppose you created a brown that has more red than the standard brown. You would convert your code as shown below.

Old Code

```
NXColor redBrown = NXChangeRedComponent(NX_COLORBROWN, 0.9);
```

New Code

```
NSColor *redBrown;
NSColor *brown;
float greenComponent, blueComponent, alphaComponent;

/* Find out the usual components of brown */
brown = [[NSColor brownColor]
     colorUsingColorSpaceName:NSCalibratedRGBColorSpace];
[brown getRed:NULL green:&greenComponent blue:&blueComponent
     alpha:&alphaComponent];

/* Create a brown with more red in it than the usual brown */
redBrown = [NSColor colorWithCalibratedRed:0.9
     green:greenComponent blue:blueComponent
     alpha:alphaComponent];
```

Color Names

NSColor objects can keep track of their own names, which means that NSColorList (formerly NXColorList) objects don't have to. NSColorList instead deals with color keys. The **allKeys** method returns an NSArray object containing all of the keys in the NSColorList. You can use the NSArray **objectAtIndex:** method to retrieve a single key from the **allKeys** array. The following table lists the methods that are now obsolete because of this change and what you should use as a replacement.

Obsolete NXColorList Method	**Possible Replacement**
colorCount TableRule.eps ¬	[[colorList allKeys] count]

generatesNamedColors TableRule.eps ¬	None necessary
localizedNameForColorNamed: TableRule.eps ¬	localizedColorNameComponent (in NSColor) localizedCatalogNameComponent (in NSColor)
nameOfColorAt: TableRule.eps ¬	[[colorList allKeys] objectAtIndex:]

Miscellaneous Obsolete Methods

In addition to the methods that handled color names, the NXColorList and NXColorWell methods in the following table are obsolete. The table shows how you can convert your code.

Obsolete Method TableHeadRule.eps ¬	**Replacement**
deactivateAllWells (NXColorWell) TableRule.eps ¬	Deactivate each color well individually.
freeAndRemoveFile (NXColorList) TableRule.eps ¬	[myColorList removeFile]; [myColorList autorelease];
saveTo: (NXColorList) TableRule.eps ¬	writeToFile:

Event Conversion

Stage 3

You can now access your application's event loop using two new classes of objects, NSRunLoop and

NSEvent. NSRunLoop defines objects that manage input sources. Applications use NSRunLoop objects to manage event loops, so you no longer have to use DPS functions to manage the event loop. An NSEvent object represents a user-generated event from the queue.

An NSRunLoop object is automatically created for each thread of the application; you generally don't need to create or to explicitly manage the run loop. Typically, the only thing you do regarding the run loop is to specify a run loop mode when retrieving events from the event queue. The run loop mode replaces the application's threshold value as shown in the following table. Like the threshold value, the run loop modes are used to block out all but events of a certain type.

For: TableHeadRule.eps ¬	**Use NSRunLoop mode:**	**To replace threshold:**
Normal operation TableRule.eps ¬	NSDefaultRunLoopMode	NX_BASETHRESHOLD
Modal panels TableRule.eps ¬	NSModalPanelRunLoopMode	NX_RUNMODALTHRESHOLD
Event-tracking loop TableRule.eps ¬	NSEventTrackingRunLoopMode	NX_MODALRESPTHRESHOLD

A fourth run loop mode, NSConnectionReplyMode is used if you're distributing objects; it lets you run Application Kit objects while the NSConnection object waits for a response.

Managing the Event Loop

You typically request events from a window rather than from the application. Windows can also post and discard events, though all of these functions are still supported by the NSApplication object too. The next three sections describe the new API for retrieving, posting, and discarding events in the event loop.

Retrieving Events

The conversion process changes methods that retrieve events from the event queue to this NSWindow method:

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

mask identifies the event type to watch for, *expiration* is an NSDate object that provides the timeout value, *mode* is one of these NSRunLoop modes, and *flag* indicates whether the event should be removed from the queue. NSWindow also defines a simpler version of this method, **nextEventMatchingMask:**, that takes only the event mask argument.

This message is to be sent to a window. In some cases, the conversion process isn't able to find a window. These cases are flagged so that you can either provide an NSWindow object or send the identical message to **NSApp**.

The conversion process flags all of the DPS functions that retrieve events. You must convert these functions to the NSWindow method or the NSApplication method yourself.

Posting Events

To post an event, you send **postEvent:atStart:** to a window or to **NSApp**. The first argument to this method is an NSEvent, which you will need to create yourself. See the NSEvent class description in the *Application Kit Reference* for a list of the methods that create an NSEvent object. The second argument takes a boolean; specify YES if you want the event posted at the start of the event queue.

Discarding Events

To discard events, you send **discardEventsMatchingMask:beforeEvent:** to a window or to **NSApp**. The first argument is a mask of event identifiers, and the second argument is an NSEvent object. This method discards all of the events matching those in the mask until it reaches the event specified by the NSEvent object. Typically, you want to send this method the last NSEvent you received.

Allowing Windows to Receive Events

Previously, if you wanted a window to be able to receive mouse-dragged events, you had to specifically enable them for that window. You did this by adding the mouse-dragged event to the mask of the events the window could receive. Now, NSWindows receive the mouse-dragged events by default. They do not have event masks. The only event that you disable and enable for a window is a mouse-moved event. You do this through the **setAcceptsMouseMovedEvents:** method.

You might also have used the window's event mask to prevent it from becoming the key window by disabling the key-up and key-down events. Now to prevent a window from becoming key, you must do one of the following:

SquareBullet.eps ¬Turn the window into an NSPanel and send it the **setBecomesKeyOnlyIfNeeded:** message with YES as the argument.
SquareBullet.eps ¬Subclass NSWindow and override the **canBecomeKeyWindow** method.

Accessing Event Information

Any information you used to receive by accessing the members of an NXEvent structure you now receive by sending messages to an NSEvent object. The conversion process will convert your code to use NSEvents instead of NXEvents. Each NXEvent field has an analogous method in NSEvent except for the character set, *event***.data.key.charSet**. The character set attribute was removed because the character code is now an NSString, and NSStrings know about their character encodings. Use the **canBeConvertedToEncoding:** method to find out the character code's character set, as shown in the following example.

Old Code

```
if (event.data.key.charSet == NX_ASCIISET) {
...
} else if (event.data.key.charSet == NX_SYMBOLSET) {
...
```

```
}
```

New Code

```
NSString *characterCode = [event characters];

if ([characterCode
     canBeConvertedToEncoding:NSASCIIStringEncoding])
     ...
else if ([characterCode
     canBeConvertedToEncoding:NSSymbolStringEncoding])
     ...
```

Using Timer Events

To create a timer event, you now use NSEvent objects, not NXTrackingTimer structures. If you use timer events, change your code as shown below. (The conversion process takes care of much of this for you.)

Old Code

```
NXTrackingTimer myTimer;
NXBeginTimer(&myTimer, 3.0, 5.0);
...
NXEndTimer(&myTimer);
```

New Code

```
[NSEvent startPeriodicEventsAfterDelay:3.0 withPeriod:5.0];
....
[NSEvent stopPeriodicEvents];
```

Keyboard Events

If your application looks for arrow keys, you will have to convert this code by hand. Arrow keys are now represented as Unicode characters rather than characters from the Symbol font encoding (codes AC through AF). The following table shows what to replace the arrow key constants with, and the example following the table shows how to change your code.

Obsolete Arrow Key TableHeadRule.eps ¬	**New Arrow Key**
NX_LEFT TableRule.eps ¬	NSLeftArrowFunctionKey
NX_RIGHT TableRule.eps ¬	NSRightArrowFunctionKey
NX_UP TableRule.eps ¬	NSUpArrowFunctionKey
NX_DOWN TableRule.eps ¬	NSDownArrowFunctionKey

Old Code

```
NXEvent *theEvent;
...
if ((theEvent->data.key.charSet == NX_SYMBOLSET) &&
     (theEvent->data.key.charCode == NX_LEFT))
     ...
```

New Code

```
NSEvent *theEvent;
NSString *eventCharacters = [theEvent characters];
...
if (![eventCharacters isEqualToString:@""] &&
     ([eventCharacters characterAtIndex:0] == NSLeftArrowFunctionKey))
     ...
```

Font Conversion

Stage 3

The Font class is now named NSFont. Like its predecessor, an NSFont object stores the attributes of a font. Because NSFont is a subclass of NSObject, it is now deallocated in the same way that other subclasses of NSObject are deallocated. Two major improvements are introduced in NSFont: an easier way to work with font metrics and automatic handling of flipped matrices.

Font Metrics

NSFont has an easier way to handle font metrics. The **metrics** and **readMetrics** methods are obsolete. Instead, NSFont provides separate methods that you can use to find out different values. For example, **advancementForGlyph:** returns the scaled dimensions of a glyph in the font. You must convert the code that reads the font metrics by hand.

Font Matrices

The flipped matrix is obsolete. Previously, if you wanted to create a font that you drew in a view, you had to flip the font using the flipped matrix so that it would appear right side up on the screen. NSFonts can correctly orient themselves with the view, so you don't have to worry about flipping the font.

Because flipped matrices are gone, the only time you need to use font matrices is if you want to perform some nonlinear function on the font. To create a normal font, you now just specify a name and a size. Most methods no longer take a matrix argument. If you do use a nonlinear matrix, you will have to change your code as summarized in the table below.

Obsolete Font Method TableHeadRule.eps ¬	**Replacement for Nonlinear Matrixes**
newFont:size:matrix: TableRule.eps ¬	fontWithName:matrix: Factor the size into the matrix.
boldSystemFontOfSize:matrix: TableRule.eps ¬	fontWithName:[[NSFont boldSystemFontOfSize:] fontName] matrix:
userFixedPitchFontOfSize:matrix: TableRule.eps ¬	fontWithName:[[NSFont userFixedPitchFontOfSize:] fontName] matrix:
userFontOfSize:matrix: TableRule.eps ¬	fontWithName:[[NSFont userFontOfSize:] fontName] matrix:
systemFontOfSize:matrix: TableRule.eps ¬	fontWithName:[[NSFont systemFontOfSize:] fontName] matrix:

Other Font Changes

The table below summarizes other changes to the Font class and to the FontManager class.

Obsolete Method TableHeadRule.eps ¬	**Possible Replacement**
name (in Font) TableRule.eps ¬	fontName

fontNum (in Font) TableRule.eps ¬	None
hasMatrix (in Font) TableRule.eps ¬	None necessary
getFamily:traits:weight:size:ofFont: (in FontManager) TableRule.eps ¬	familyName pointSize traitsOfFont: weightOfFont:

Icon and Image Conversions

Stage 2

The differences between using images and icons in NEXTSTEP Release 3 and using images and icons in OpenStep are relatively minor. This section summarizes the changes to images and icons, focusing on the changes you might need to perform yourself.

Referring to Images and Icons in Code

The major change to image and icon support is that you now refer to icons in your application as NSImages. You no longer use the icon name. This change is shown in the following example.

Old Code

```
[appButton setIcon: iconName];
```

New Code

```
[appButton setImage:[NSImage
```

```
    imageNamed:[NSString stringWithCString:iconName]]];
```

Initializing Scalable Images

NXImage had one initialization method that used the contents of a file, **initFromFile:**. NSImage has two such methods: **initWithContentsOfFile:** and **initByReferencingFile:**. For scalable images, the correct method to use is **initByReferencingFile:** because it assumes that the file will persist and can be referenced later if the image is resized. The conversion process examines your NXImages and tries to choose the correct initialization method, but you should verify that it has chosen the right one.

Swapping Functionality Between Classes

Some of the functionality that used to be in one of the NSImage or NSImageRep classes has been moved to another class.

SquareBullet.eps ¬You now register NSImageRep subclasses with the NSImageRep class object.

SquareBullet.eps ¬Retrieving information about a TIFF file has been moved to the NSBitmapImageRep class.

Other Changes

The following are other changes to NSImage methods that you may have to make yourself. The conversion process takes care of all other changes.

SquareBullet.eps ¬The **initFromSection:** method is obsolete and is replaced with **imageNamed:**. The conversion process makes this change and uses the argument you passed to **initFromSection:**, which identified the section, as the image's name. You may want to change this.

SquareBullet.eps ¬The **lastRepresentation** method is obsolete. Instead, you can use the message **[[*myImage* representations] lastObject]**. The **representations** method returns an NSArray. As before, though, if multiple representations are added at the same time, you can't predict which image will be returned by **lastObject**.

SquareBullet.eps ¬The **lockFocus** method now raises an exception to indicate an error rather than returning a **BOOL**. If you used to check the return type of **lockFocus** to make sure there were no errors, you should now use **isValid** instead.

Icon and Image Gotchas

After the icon and image conversions, look out for the following:

Highlighted Buttons

Previously, when a button contained an image with a gray background, the entire image was highlighted when the button was highlighted. In the NeXT implementation of OpenStep, the gray background of the image will be highlighted only if the image has alpha.

initWithSize: in NSImage

If you're converting in stages and you use the **initWithSize:** method to initialize an NSImage, you may receive a compiler error after stage 2. The **initWithSize:** method now takes an NSSize structure, so it conflicts with NXData's **initWithSize:** method, which takes an unsigned integer as an argument. Because the NSSize structure is larger than an unsigned integer, this statement produces an error. To avoid the error message so that you can continue with the conversion, put the **alloc** and **initWithSize:** methods in separate statements as shown in the following example. NXData is obsolete, and it will be converted in a later stage.

Old Code

```
NXImage *foo;
NSSize mySize;
foo = [[NXImage alloc] initSize:&mySize];
```

New Code

```
NSImage *foo;
```

```
NSSize mySize;

foo = [NSImage alloc];
foo = [foo initWithSize:mySize];
```

imageNamed: Methods

You can no longer obtain icons that are in a MachO segment with the **imageNamed:** method. Instead, you should store icons in the application's wrapper. If your application accesses a named image that it cannot find but the image exists in the MachO section, this will be logged at run time. If you receive such a run-time error, move the icon into the application wrapper.

Creating Multi-Image TIFFS

If your code creates multi-image TIFFs, you can now use the NSImageRep method **TIFFRepresentationOfImageRepsInArray:** to do this. For more information, see the *Application Kit Reference*.

Matrix And Cell Conversion

Stage 5

The changes to Controls and Cells in OpenStep are very minor; not much about the existing functionality has changed. There are some new features, however, and you might want to take advantage of them.

SquareBullet.eps ¬You can attach any object to a cell with the method **setRepresentedObject:**. This method saves you from having to subclass a cell in many cases.

SquareBullet.eps ¬An NSMatrix can sort its cells.

SquareBullet.eps ¬Keyboard equivalents for button cells can use any modifier key, not just the Command key.

For more information, see the *Application Kit Reference*. The rest of this section tells you the changes to Controls and Cells you may have to make yourself.

Modifying a Cell

Matrix and Form objects used to be able to modify their cell's tags, targets, actions, and values. Now NSMatrix and NSForm can modify only the cell's physical attributes and state. To modify a cell's tag, target, action, or value, you must first retrieve the cell and then modify it.

Old Code

```
const char *empName = [aForm stringValueAt:0];
```

New Code

```
NSString *empName = [[aForm cellAtIndex:0] stringValue];
```

Overriding the Cell State

Previously, you could send a Matrix the **setReaction:NO** message so that a cell would not change states if the cell's action method deleted the cell or completely changed the physical appearance of the cell. This is no longer necessary with the changes to NSView. The method in the following table is obsolete.

Obsolete Matrix Method TableHeadRule.eps ¬	Possible Replacement
setReaction: TableRule.eps ¬	None necessary

PopUp Conversion

Stage 5

Previously, the Application Kit provided a PopUpList class that defined pop-up and pull-down lists. Both types of lists were controlled by a trigger button whose target was the PopUpList object. When you created a list in Interface Builder, you actually created the button. To access the PopUpList, you sent the button a **target** message. The new Application Kit has simplified the API for pop-up and pull-down lists. You now maintain a single object, NSPopUpButton, which defines both the trigger button and the pop-up list. The conversion process changes your PopUpList objects to NSPopUpButtons.

Accessing Items in the Matrix

PopUpList had a Matrix object containing MenuCell objects, which represented the items in the list. To access individual items in the list, you first retrieved the matrix with **itemList** and then used Matrix methods to access the items.

NSPopUpButton has methods that allow you to access the items in the list directly. The **itemList** method still exists (renamed **itemArray**), but you shouldn't have to use this array to access individual items in the list.

Old Code

```
int index = [[imagePopUp itemList] selectedRow];
```

New Code

```
int index = [imagePopUp indexOfSelectedItem];
```

Targets and Actions of NSPopUpButtons

You typically gained access to a PopUpList object by invoking the **target** method on its trigger Button object. Now that there a a single object defining both the list and the button, **target** returns the target of that object. In most cases, you can remove the **target** message and set the NSPopUpButton's target in Interface Builder. (See the chapter ªDebugging Tipsº for instructions on converting your nib file.)

Also, the NSPopUpButton sends its action message directly to the target, whereas previously the sender of PopUpList's action message was its Matrix. You will have to inspect your action method to make sure you are sending appropriate messages to the new sender, NSPopUpButton.

Obsolete Items

The methods and functions shown in the following table are now obsolete because of the changes to the way pop-up lists and pull-down lists are created.

Obsolete Item	**Possible Replacement**
TableHeadRule.eps ¬	
count method TableRule.eps ¬	numberOfItems method
getButtonFrame: method TableRule.eps ¬	frame method
NXAttachPopUpList() TableRule.eps ¬	Create an NSPopUpButton instance. The list is automatically attached.
NXCreatePopUpListButton() TableRule.eps ¬	Create an NSPopUpButton instance. The list is automatically attached.
popUp: TableRule.eps ¬	None needed. This action is now automatic.

Printing Conversion

Stage 4

The Application Kit has a new class to handle printing, NSPrintOperation. NSPrintOperation objects represent print jobs and EPS code-generation operations. Whenever a user prints a document or copies EPS data, an NSPrintOperation object is created to oversee the generation of PostScript® code. When the operation is complete, the object is released.

Previously, applications (except document-oriented applications) had a shared PrintInfo object that kept track of everything about a print operation: the status of the operation, the job-specific attributes, and the page layout attributes. Whenever a new print job was requested, the attributes stored in the shared PrintInfo had to be reset to their default values.

Now, the shared NSPrintInfo holds only the page layout attributes. The NSPrintOperation runs the Print panel and keeps track of the operation's status. The NSPrintOperation takes the page layout information from the shared NSPrintInfo and the job-specific information from the Print panel to create a new NSPrintInfo object that contains all of the attributes specific to this print operation. In this way, the shared NSPrintInfo never has to be reset. The NSPrintOperation does not itself perform the printing; instead, it sends a message to the NSView object.

The following figure shows what happens now during a typical print operation. Almost all of the changes to the printing mechanism take place automatically. You have to rewrite code only if your application tries to control a print operation.

PrintingModel.eps ¬

Accessing the Shared NSPrintInfo

NSPrintInfo has a new method, **sharedPrintInfo**, which replaces Application's **printInfo** method. The conversion process make this change for you. However, the shared NSPrintInfo now contains only the page layout attributes that will be applied to every print operation; it does not contain any other attributes. Because of this change, the conversion process flags each occurrence of the message **[NSPrintInfo sharedPrintInfo]** to make sure that you truly want the shared NSPrintInfo object. To access an NSPrintInfo object, you have the choices shown in the following table.

Use This Method: **TableHeadRule.eps ¬**	**If You Want To:**
sharedPrintInfo (in NSPrintInfo) TableRule.eps ¬	Find out or change the default page layout attributes for the application.
printInfo (in NSPrintOperation) TableRule.eps ¬	Find out the attributes being used for a specific operation.

Changes to NSPrintInfo Methods

To improve the API, the NSPrintInfo methods in the following table have changed as shown.

Obsolete PrintInfo Method **TableHeadRule.eps ¬**	**Possible Replacement**	**Notes**
setPaperRect: TableRule.eps ¬	setPaperSize:	Change the type of the argument to NSSize.
getMarginLeft:right:top:bottom:	leftMargin rightMargin topMargin bottomMargin	You may be able to delete some code.

TableRule.eps ¬

setMarginLeft:right:top:bottom:	setLeftMargin: setRightMargin: setTopMargin: setBottomMargin:	You may be able to delete some code.

TableRule.eps ¬

For example, to set the left and right margins to 50 and leave the top and bottom margins alone, you used to have to retrieve all four margins first. Now all you have to do is set the left and right margins.

Old Code

```
float left, right, top, bottom;

[printInfo getMarginLeft:&left right:&right top:&top
     bottom:&bottom];
[printInfo setMarginLeft:50 right:50 top:top bottom:bottom];
```

New Code

```
[printInfo setLeftMargin:50];
[printInfo setRightmargin:50];
```

Changing Print-Job Attributes in NSPrintInfo

NSPrintInfo no longer defines methods that change print-job attributes because typically only NSPrintOperation objects set them. NSPrintInfo still stores print-job attributes, but now these attributes and all other printing attributes are stored in an NSMutableDictionary rather than in separate instance variables. Use the **dictionary** method to access this dictionary, and use **setObject:forKey:** to change a value stored in the

dictionary. The file **NSPrintInfo.h** lists all of the possible dictionary keys.

The following example shows how to convert code that changes the scaling attribute.

Old Code

```
[myPrintInfo setScalingFactor:1.4];
```

New Code

```
[[myPrintInfo dictionary]
     setObject:[NSNumber numberWithFloat:1.4]
     forKey:NSPrintScalingFactor];
```

Subclassing NSPrintInfo

If you subclass NSPrintInfo, you might need to do the following:

SquareBullet.eps ¬To add new attributes to an NSPrintInfo, you can add them to its dictionary. Use **dictionary** to retrieve the dictionary, and send it **setObject:forKey:** to add your attributes.

SquareBullet.eps ¬Override the method **setUpPrintOperationDefaultValues**. NSPrintOperation invokes this method when the print operation starts so that it can set the default values for attributes that aren't set on the Print panel or the Page Layout panel. If you subclass NSPrintInfo, you may need to override this method to set default values for custom keys.

Creating an NSPrintOperation

In any method that tries to control a printing operation or generation of EPS code, you need to do the following:

1. If the operation is not going to use the shared NSPrintInfo, create an NSPrintInfo object and set the appropriate attributes.
2. Create an NSPrintOperation object to oversee the operation. The method you use to do this depends upon what type of operation you need to perform (EPS code generation or printing) and whether the operation is to use the shared NSPrintInfo. See the *Application Kit Reference* for a complete list.
3. Start the operation by sending the message **runOperation**. This method invokes the appropriate NSView method.

For example, suppose one of the objects in your application implements a method that is the target for the Print command. You would convert it as shown below.

Old Code

```
- print:sender
{
     if (document && ![document isEmpty]) {
          [NXApp setPrintInfo:[document printInfo]];
          [[document view] printPSCode:self];
     }
     ...
     return self;
}
```

New Code

```
- print:sender
{
     BOOL printSuccess;
     if (document && ![[document view] isEmpty]) {
          printSuccess = [[NSPrintOperation
```

```
            printOperationWithView:[document view]
            printInfo:[document printInfo]]
            runOperation];
    }
    ...
    return self;
}
```

PrintPanel

There is no PrintPanel class in the OpenStep specification. It's NSPrintOperation's responsibility to run the standard Print panel and to retrieve job-specific information from it. In OpenStep, if you want to change the look of the Print panel, you send **setAccessoryView:** to the NSPrintOperation. The methods **updateFromPrintInfo** and **finalWritePrintInfo** are sent to this accessory view.

NeXT provides an NSPrintPanel object as an extension to the specification. Other than the removal of the three methods mentioned here, this class's API is similar to the Release 3 PrintPanel's API.

Printing Windows

The Window class defined numerous printing methods. Almost all of these methods have been removed because they duplicate functionality in NSView. As a result, the API is simpler, but there is less flexibility in printing a window. If you really need the flexibility, you can create an EPS version of the window, place it in a view, and print the view. The printing methods shown in the following table remain in NSWindow.

Obsolete Window Method	**New Name in NSWindow**
TableHeadRule.eps ¬	
smartPrintPSCode:	print:
TableRule.eps ¬	

copyPSCodeInside:to: TableRule.eps ¬	dataWithEPSInsideRect:

Printing Views

The View method **printPSCode:** has been renamed **print:**. If you define a **print:** method for an object, you might receive duplicate definition warnings at compile time. Because of this, the conversion process flags all of the **print:** methods in your code and asks you if you want to rename them to avoid the duplicate definitions warning.

Similarly, the **faxPSCode:** method has been renamed **fax:**. (This method is a NeXT extension to the OpenStep specification. OpenStep does not support faxing.) The **faxPSCode:toList:numberList:sendAt:wantsCover:wantsNotify:wantsHires:faxName:** method has been removed from the NSView class. Instead of using this method, set up an NSPrintInfo with the attributes you used to specify in this method, and create an NSPrintOperation to perform the faxing. The conversion process makes this change for you.

The following table summarizes the changes to the printing and faxing methods.

Obsolete View Method **TableHeadRule.eps ¬**	**Possible Replacement**
printPSCode: TableRule.eps ¬	print:
faxPSCode: TableRule.eps ¬	fax: (NeXT extension)
faxPSCode:... TableRule.eps ¬	fax: (NeXT extension)

Overriding Printing Methods in NSView Subclasses

Some methods in the View API were invoked by the **printPSCode:** method. You might have subclassed View to override these methods and create a custom print operation. These methods are obsolete because the NSPrintOperation object sets up the print operation now. If you overrode any of the methods shown in the following table, you must now subclass NSPrintOperation and override the corresponding methods. The **print:** method in your NSView subclass should use your NSPrintOperation subclass.

Obsolete View Method TableHeadRule.eps ¬	**Possible Replacement**
beginPSOutput TableRule.eps ¬	Override createContext (in NSPrintOperation).
endPSOutput TableRule.eps ¬	Override deliverResult (in NSPrintOperation).
openSpoolFile: TableRule.eps ¬	Override createContext (in NSPrintOperation).
spoolFile TableRule.eps ¬	Override deliverResult (in NSPrintOperation).

Printing Gotchas

After the Printing conversion, look out for the following:

NXDrawingState

The NSDPSContext method **isDrawingToScreen** replaces the **NXDrawingStatus** global variable. (There is also

an **isEPSOperation** method in NSPrintOperation.) In this way, the drawing context is switched with every context switch. The conversion process changes the uses of the **NXDrawingStatus** variable for you, but you should delete any place where you changed its value.

Be certain to check that the current context is not **nil** before you send **isDrawingToScreen**. If the **currentContext** is **nil**, you will receive NO if you send the message **[[NSDPSContext currentContext] isDrawingToScreen]**.

NXMeasurementUnit

The **NXMeasurementUnit** enum previously defined in **PageLayout.h** is obsolete. If you want to find out the user's choice for the units that the page size is specified in, check the **NSMeasurementUnit** user default in the **NSGlobalDomain**. For more information on user defaults in OpenStep, see the section ªDefaults Conversionº in the chapter ªConverting the Common Classes.º

NXPrintingUserInterface Protocol

The NXPrintingUserInterface protocol is obsolete. Instead of implementing this protocol, use the NSPrintOperation **setShowPanels:** method.

Screen Conversion

Stage 4

In OpenStep, screens are represented by NSScreen objects rather than NXScreen structures, and screens are no longer identified by screen numbers. This affects the Window methods that place windows on specific screens. Instead of specifying an NXScreen structure for these methods, you use the NSScreen method **frame** to retrieve the screen's size and location, then place the window inside that frame. The following table lists the obsolete Window methods and shows what to use as a replacement.

Obsolete Window Method TableHeadRule.eps ¬	Possible Replacement
getFrame:andScreen: TableRule.eps ¬	screen, frame
moveTo::screen: TableRule.eps ¬	setFrame:display:
moveTopLeftTo:screen: TableRule.eps ¬	setFrame:display:
placeWindow:screen: TableRule.eps ¬	setFrame:display:

Spell Checker Conversion

Stage 2

Previously, you performed spell checking on any object that conformed to the NXReadOnlyTextStream and NXSelectText protocols. In OpenStep, you can perform spell checking on any NSString. The biggest change to spell checking is that the spell-checking methods no longer select the misspelled word in the text and display it in the Spelling panel; they now just return the range of the first misspelled word they find. If you want the text selected and the misspelled word displayed, you must write this code yourself using methods from the NSSpellChecker class. The following table summarizes the protocols, types, and methods that are obsolete due to changes in the spell-checking scheme. For more information, see the NSSpellChecker class specification in the *Application Kit Reference*.

Obsolete Item TableHeadRule.eps ¬	Replacement
NXReadOnlyTextStream protocol TableRule.eps ¬	You can now perform spell checking on any NSString.

NXSelectText protocol TableRule.eps ¬	You can now perform spell checking on any NSString.
NXSpellCheckMode type TableRule.eps ¬	Spell-checking methods always stop at the first misspelled word.
addGuess TableRule.eps ¬	The spellServer:suggestGuessesForWord:inLanguage: delegate method returns the guesses.

Text Conversion

Stage 2

In OpenStep, there is a new Text object, NSText, with a much simpler API. NSText uses NSStrings, so it allows for internationalization. The conversion process changes all instances of Text in your code to NSText.

In simplifying the API, some functionality was removed from NSText. You can still use RTF and RTFD formats, import graphics, and manage the font and alignment, so this won't affect most applications. However, if you need to perform some specialized operations (such as using your own drawing function, managing the text runs, or using different paragraph styles within a single object) or if you need to create a subclass, you should use NSCStringText instead. NSCStringText is simply the old Text object renamed to remind you that it understands only 8-bit character encodings and can't be fully internationalized.

This section describes the changes to the Text object that you need to know about during the conversion. In some cases, you may have to perform some conversion yourself.

Text Length, Character Positions, and Ranges of Characters

NSText has a new method, **string**, which returns its contents in an NSString. Many of the methods that Text used to implement are replaced with methods sent to the NSString returned by **string**. The following table lists the obsolete methods and shows possible replacements for them.

Obsolete Text Method TableHeadRule.eps ¬	Possible Replacement
byteLength TableRule.eps ¬	[[*myText* string] cStringLength]
charLength TableRule.eps ¬	[[*myText* string] length]
getSubstring:start:length: TableRule.eps ¬	[[*myText* string] substringWithRange:]
textLength TableRule.eps ¬	[[*myText* string] length]
offsetFromPosition: TableRule.eps ¬	[[*myText* string] rangeOfComposedCharacterSequenceAtIndex:]
positionFromOffset: TableRule.eps ¬	[[*myText* string] rangeOfComposedCharacterSequenceAtIndex:]

Note: The conversion between **byteLength** and **cStringLength** is not a one-to-one correspondence. If the text in your object cannot be represented as a C string, **cStringLength** returns 0. Consider changing **byteLength** to something else.

Retrieving the Selection Range

Previously, the method **getSel::** returned two selection points, which you could use to retrieve information about the range of the selected text. Although this method is still implemented in NSCStringText (renamed **getSelectionStart:end:**), it is not implemented in NSText. To perform a similar function, use **selectedRange** to receive an NSRange that indicates the starting point and the length of the range. You can use the **NSMaxRange()** function to determine the ending point of the selection if necessary. The following example illustrates this change.

Old Code

```
- changeSelectionToPlatypus:sender
```

```
{
/* This method changes the selected text to "Platypus" only
€€€if doing so will not increase the length of the text. */

     NXSelPt start, end;
     int allowedLength = strlen("Platypus");

     [myText getSel:&start:&end];
     if ((end.cp - start.cp + 1) >= allowedLength)
         [myText replaceSel:"Platypus"];

    return self;
}
```

New Code

```
-(void)changeSelectionToPlatypus:sender
{
    NSRange selectedRange;
    int allowedLength = [@"Platypus" length];

    selectedRange = [myText selectedRange];
    if (selectedRange.length >= allowedLength)
        [myText setCharactersInRange:selectedRange
           withString:@"Platypus"];

}
```

Using RTF and RTFD Formatted Text

All of the methods that used to read and write RTF and RTFD text from and to a stream are obsolete. They have been condensed into four methods to read and write RTF and RTFD text. The following table lists the obsolete methods and shows which method you should use as a replacement. As stated previously, in some cases you may need to change the declaration of a variable from a stream to an NSData object.

Obsolete Text Method TableHeadRule.eps ¬	**Replacement**
readRichText: TableRule.eps ¬	replaceCharactersInRange:withRTF: (last argument is NSData object)
readRichText:atPosition: TableRule.eps ¬	replaceCharactersInRange:withRTF:
readRTFDFrom: TableRule.eps ¬	replaceCharactersInRange:withRTFD: (last argument is NSData object)
replaceSelWithRichText: TableRule.eps ¬	replaceCharactersInRange:withRTF:
replaceSelWithRTFD: TableRule.eps ¬	replaceCharactersInRange:withRTFD:
writeRichText: TableRule.eps ¬	RTFFromRange: (returns NSData)
writeRichText:from:to: TableRule.eps ¬	RTFFromRange:
writeRTFDTo: TableRule.eps ¬	RTFDFromRange: (returns NSData)
writeRTFDSelectionTo: TableRule.eps ¬	RTFDFromRange:
readText: TableRule.eps ¬	setString:

In addition to these changes, the NXRTFDError enumerated type and the NXRTFDErrorHandler protocol are obsolete. Methods that used to return an NXRTFDError now return **BOOL**. The **attemptOverwrite:** method defined in the protocol is no longer used when writing RTFD data to a file.

Text Gotchas

After the Text conversion, look out for the following:

string Method

The **string** method returns a pointer to the contents of the NSText, meaning the value inside the returned NSString changes as the NSText changes. For example, suppose you have an NSText object that contains the string ªNEXTSTEPº and you assign a temporary variable **myString** to the NSString returned by that NSText's **string** method. **myString** contains ªNEXTSTEP.º Now suppose the user changes the contents of the NSText to ªOPENSTEP.º **myString**'s value is now ªOPENSTEPº as well.

If you want a snapshot of the NSText's contents rather than a pointer to the NSText's contents, send the **copy** message to the NSString returned by **string**.

textWillEnd:

The delegate method **textWillEnd:** is replaced by **textShouldEndEditing:**. Like **textWillEnd:**, **textShouldEndEditing:** is invoked when the NSText object is about to give up first responder status. However, the return value is interpreted differently. **textWillEnd:** returned YES to prevent the Text object from giving up first responder status. **textShouldEndEditing:** returns YES to allow the NSText object to give up first responder status. Examine the values your delegate method is returning very closely.

NXCharFilterFunc

If you implemented a character filter function, you must change it manually. (Character filter functions are valid only for NSCStringText.) The new template for character filter functions is:

```
typedef€unsigned€short (*NSCharFilterFunc)
     (unsigned€short€charCode, int€flags,
     NSStringEncoding€theEncoding);
```

NSStringEncoding is an enumerated type that identifies the character encoding used on your system. You may want to use **[NSString defaultCStringEncoding]** (which returns the default encoding for your system) for this parameter.

View Conversion

Stage 4

The API for Views has been simplified in many areas, making many of the existing View methods obsolete. Most of the changes in this area involve removing View and Window methods that are no longer necessary. If you use an obsolete View method, the conversion process changes it for you. If you subclass View and override an obsolete method, the conversion process flags it, and you need to either change the implementation or delete the method. This section describes the View conversions that you may need to perform yourself.

Drawing Views

This section describes changes to View drawing methods.

Clipping

To simplify drawing API, NSViews cannot draw outside their bounds; they always clip to their bounds

rectangle, which makes methods in the following table obsolete. Use the replacements listed in the table.

Obsolete View Method TableHeadRule.eps ¬	Possible Replacement
display:andClip: TableRule.eps ¬	displayRect:
clipToFrame: TableRule.eps ¬	For nonrectangular views, use display to draw the view. Override isOpaque: so that it always returns NO.

Opaqueness

Previously, you could send the message **displayFromOpaqueAncestor:::** to redraw a transparent view using its first opaque ancestor. The transparent view would go up its view hierarchy until it found an opaque view, and then each view from the opaque view down to the transparent view in question would be redrawn.

In OpenStep, this is the default behavior; all transparent views use their first opaque ancestors to redraw themselves. If you have a transparent view that doesn't need to use its first opaque ancestor, send it the message **displayIfNeededIgnoringOpacity**.

Previously, View provided the **setOpaque:** method, which set whether it was opaque or not. You really could use this method only in subclasses of view; other objects could not send **setOpaque:** to a view to request that it become opaque. To clean up the API, this method is now obsolete. If you sent **setOpaque:** to the Text object to eliminate the background gray, change it to **setDrawsBackground:NO**. If you subclassed View, override the **isOpaque** method to return whether your view is opaque or not. The default is that the view is not opaque (is transparent). Because transparent views are redrawn from their first opaque ancestor, they take longer to redraw. For this reason, it's important that you override **isOpaque** to return YES if your NSView subclass is opaque.

Obsolete View Method TableHeadRule.eps ¬	Possible Replacement
displayFromOpaqueAncestor::: TableRule.eps ¬	display
setOpaque: TableRule.eps ¬	setDrawsBackground:NO if sent to Text Override isOpaque to return YES.

Focusing

To make the API more clear, all focusing methods have been moved to NSView. The following methods are obsolete.

Obsolete Method TableHeadRule.eps ¬	Possible Replacement
focusView (in Application) TableRule.eps ¬	focusView (NSView class method)
isFocusView (in View) TableRule.eps ¬	[NSView focusView] == myView If you override isFocusView, remove your implementation.

Drawing Optimizations

Some rarely used methods that performed the same operations as some PostScript functions did are obsolete. The following table lists these functions and what to use as a replacement.

Obsolete View Method TableHeadRule.eps ¬	Possible Replacement
drawInSuperview TableRule.eps ¬	allocateGstate
setClipping:, doesClip TableRule.eps ¬	allocateGstate

Flipped Coordinate Systems

Previously, View provided the **setFlipped:** method, which set whether the view used a flipped coordinate system. You really could use this method only in subclasses of view; another object could not use **setFlipped:** to request that a view's coordinate system become flipped. To clean up the API, this method and other methods that handled flipped coordinate systems are obsolete. The following table lists the obsolete methods and what to use as a replacement. Note that you can still flip an NSImage.

Obsolete View Method TableHeadRule.eps ¬	Possible Replacement
descendantFlipped: TableRule.eps ¬	Not needed. The flipped property cannot change.
notifyWhenFlipped: TableRule.eps ¬	isFlipped
setFlipped:	None. If you override this method, instead override isFlipped: to always return YES. TableRule.eps ¬

Accepting Mouse Events

The **acceptsFirstMouse** method now receives an NSEvent object representing the mouse event in question. You can now base the return value of this method on the location of the mouse.

Miscellaneous Obsolete Methods

In addition to the methods mentioned previously, the following methods have been removed from NSView to make the API cleaner and simpler.

Obsolete View Method	**Replacement**
convertPointToSuperview TableHeadRule.eps ¬	convertPoint:toView:
convertPointFromSuperview TableRule.eps ¬	convertPoint:fromView:
convertRectToSuperview TableRule.eps ¬	convertRect:toView:
convertRectFromSuperview TableRule.eps ¬	convertRect:fromView:
findAncestorSharedWith: TableRule.eps ¬	None.
notifyToInitGstate TableRule.eps ¬	None necessary. Views are always given a chance to initialize their gstates.

Automatically Updating the Display

In OpenStep, the mechanism that automatically updates the display has been greatly simplified. When you make a change that affects a view's appearance, the following sequence of events occurs:

1. When a view needs to be redrawn, you set its **needsDisplay** flag. If the entire view needs redrawing, use **setNeedsDisplay:**. If only a portion of the view needs redrawing, use **setNeedsDisplayInRect:**.
2. The **setNeedsDisplay...** message posts an asynchronous notification. The view's window receives this notification after the current user event has completed.
3. The window sends it border view a **displayIfNeeded** message. The border view checks all of the views in the view hierarchy. If a view's **needsDisplay** flag is set, the portion of the view that needs to be redrawn is redrawn.

You can disable this automatic updating of the display with NSWindow's **setAutodisplay:** method. When this mechanism is disabled, the **setNeedsDisplay...** message neither sets the **needsDisplay** flag nor posts the notification. If autodisplay is disabled, you can still use the NSView's **display** method to redraw the view. You can also use the **display** method any time you want to redraw the view immediately without waiting for the event to complete.

Previously, there were a number of mechanisms that updated the display, and many different classes provided various degrees of control over them. All of these mechanisms have been condensed into the one described here, which is much easier to use. There is only one way to have the display automatically updated, and there is only one way to turn it off. The following table lists the Application, View, and Window methods that are obsolete because of this change and what you should use as a replacement.

Obsolete Method TableHeadRule.eps ¬	**Possible Replacement**
setAutoupdate: (in Application) TableRule.eps ¬	setAutodisplay: (in NSWindow)
disableDisplay (in Window) TableRule.eps ¬	None
enableDisplay (in Window) TableRule.eps ¬	None

isDisplayEnabled (in Window) TableRule.eps ¬	None
invalidate:: (in View) TableRule.eps ¬	setNeedsDisplay:YES
isAutodisplay (in View) TableRule.eps ¬	None
setAutodisplay: (in View) TableRule.eps ¬	None (use display directly)
setsDisplayOnScroll: (in ClipView) TableRule.eps ¬	None
update (in View) TableRule.eps ¬	displayIfNeeded, displayIfNeededInRect:

Similarly, the scheme for automatically enabling and disabling menu commands has changed. Previously, if you wanted a menu cell to be automatically updated, you implemented a method that determined whether the menu cell should be enabled or disabled, and you sent the name of that method to the MenuCell in a **setUpdateAction:forMenu:** message.

In OpenStep, all menu items are automatically updated by default. The scheme for enabling and disabling menus is simplified. For each item in the menu, the NSMenu object looks for an object in the responder chain that responds to the item's action message. If it finds an object that responds to the message, the item is enabled. If not, the item is disabled.

In many cases, this menu updating scheme is sufficient, and you can delete your code that updated menu items. In a few instances, you may still need to control whether an item is enabled or disabled yourself. For example, if your object implements the item's action method but the method should only be invoked if a certain type of item is selected, you can implement the method **validateMenuItem:** in that object to disable the item. Before NSMenu enables the item, it looks to see if the object that implements the action method also implements **validateMenuItem:**. If so, it invokes that method to determine whether it should enable the item. **validateMenuItem:** is defined in the NSMenuValidation informal protocol. See its specification sheet in the

Application Kit Reference for more information. The following table summarizes the effect this change has had on the MenuCell methods for enabling and disabling a menu cell.

Obsolete MenuCell Method TableHeadRule.eps ¬	Possible Replacement
setUpdateAction:forMenu: TableRule.eps ¬	None necessary.
updateAction TableRule.eps ¬	None necessary. The updateAction, if it exists, is always validateMenuItem:.

View Gotchas

After the View conversion, look out for the following:

Redrawing Transparent Views

By default, OpenStep views return NO for **isOpaque**, meaning that by default, OpenStep views are transparent. Redrawing transparent views is much slower than redrawing opaque views. When a transparent view needs to be redrawn (that is, its **needsDisplay** flag is set), the first opaque ancestor of the view must be redrawn as well as the view itself and all of the views in between.

If your view is opaque, be sure you are overriding **isOpaque** to return YES. The automatic display-updating mechanism uses **isOpaque** to check whether to redraw the superview as well as the view itself. If all of your opaque views return YES for **isOpaque**, display updating will be much faster.

Releasing Views

Superviews **retain** all subviews as they are added to the hierarchy and **release** them as they are removed. If you swap views in and out of the hierarchy, you should **retain** the views that are not in the hierarchy (and **release** them after you add them to the hierarchy).

Previously, sending the **free** message to a view had the side effect of removing it from the view hierarchy. This message is converted to a **release** message, which does not remove the view from the hierarchy because the superview has retained it. To get the same effect, you must first remove the view from the hierarchy and then release it.

Window Conversion

Stage 4

Like NSView, NSWindow is essentially the same as its NEXTSTEP Release 3 counterpart. However, many portions of its API have been simplified, so you may find yourself deleting many messages. This section describes the parts of the Window conversion that you may have to perform yourself.

Represented Filenames

Previously, you could use **setTitleAsFilename:** to set the title of the window to the name of the file that appears inside the window. In OpenStep, you can now set the name of the file that the window represents independent of having it appear in the titlebar. This allows you to have the document dragging icon (which you get if you Alternate-drag from the miniaturize button) represent something meaningful without having to change the name of the window. **setRepresentedFilename:** sets the file that the window represents, but it does not change the window's title. **setTitleWithRepresentedFilename:** uses **setRepresentedFilename:** to set the represented file and also changes the title of the window.

Obsolete Window Method	**Replacement**
TableHeadRule.eps ¬	
setTitleAsFilename:	setTitleWithRepresentedFilename:
TableRule.eps ¬	

Tracking Rectangles

The API for tracking rectangles has been simplified to allow greater window system independence. You now create and delete tracking rectangles in an NSView object. The tag is returned when you create a tracking rectangle, so you no longer have to create your own tags.

The following table lists the tracking rectangle Window methods that are now obsolete and what their replacements are in NSView.

Obsolete Window Method TableHeadRule.eps ¬	Possible Replacement
discardTrackingRect: TableRule.eps ¬	removeTrackingRect: (in NSView)
setTrackingRect:inside:owner:tag:left:right: TableRule.eps ¬	addTrackingRect:owner:assumeInside: (in NSView)

Style Masks

To simplify specifying a window's style, the window button mask has been merged with the style mask. In addition, several window styles that were rarely used are now obsolete. The following table summarizes the changes.

Obsolete Window Items TableHeadRule.eps ¬	Possible Replacement
buttonMask method	styleMask method

TableRule.eps ¬	
NX_MENUSTYLE mask	None
TableRule.eps ¬	
NX_MINIWINDOWSTYLE mask	None
TableRule.eps ¬	
NX_MINIWORLDSTYLE mask	None
TableRule.eps ¬	
NX_TOKENSTYLE mask	None
TableRule.eps ¬	
NX_FIRSTWINSTYLE mask	None
TableRule.eps ¬	
NX_LASTWINSTYLE mask	None
TableRule.eps ¬	
NX_NUMWINSTYLES mask	None
TableRule.eps ¬	

Miscellaneous Obsolete Window Methods

The following table lists methods that have been removed because they were rarely used and what you can use as a replacement.

Obsolete Window Method	**Possible Replacement**
TableHeadRule.eps ¬	
counterpart	miniWindowImage
TableRule.eps ¬	
addCursorRect:forView:	addCursorRect (in NSView)
TableRule.eps ¬	
removeCursorRect:forView:	removeCursorRect (in NSView)
TableRule.eps ¬	

avoidsActivation TableRule.eps ¬	None
setAvoidsActivations: TableRule.eps ¬	None

Releasing Windows

Windows created in Interface Builder are not released until the user quits the application. If you want a window to be released when the user closes it, you can do one of the following:

SquareBullet.eps ¬ Set the ªRelease when closedº attribute in Interface Builder.
SquareBullet.eps ¬ Send the window a **setReleasedWhenClosed:YES** message in your code.
SquareBullet.eps ¬ Have the delegate release the window in its **windowShouldClose:** method.