OpenStep Journal, Spring 1995 (Volume 1, Issue 1).

The Same, Yet Different: NEXTSTEP and OpenStep

Written by **Jean Ostrem**

Release 4.0 will be the first fully OpenStep™-compliant release of NEXTSTEP™. Because of some important differences between OpenStep and previous versions of NEXTSTEP, existing applications will have to be upgraded before they will run under Release 4.0. Conversion tools provided with the release will make this upgrading easier. By understanding the changes that OpenStep will bring, developers can modify their current application designs and programming projects to make conversion
easier later.

OPENSTEP TO THE RESCUE

OpenStep, as you probably know by now, is an application programming interface (API) based on NeXT™Computer's open object layer. Using OpenStep simplifies and speeds the process of constructing complex and portable client/server software that can run on multiple platforms by providing a framework for distributed computing.

This article frequently uses the term OpenStep *to mean both NEXTSTEP 4.0 and any OpenStep implementation.*

In other words, it helps you develop real corporatewide applications faster than you ever could before. It may not make developing applications a piece of cake, but it should allow you to focus more on creating the unique pieces of your application and less on reinventing common mechanisms over and over again.

NEXTSTEP Release 4.0 will be the first OpenStep-compliant release of NEXTSTEP. Although applications will look to users much the same as they did in previous releases, Release 4.0 applications will have some important differences. This means that to realize all of the benefits of OpenStep that come with Release 4.0, you'll first have to upgrade existing applications. Conversion tools will be provided with Release 4.0, and they'll simplify the process of moving applications to OpenStep. In the meantime, you can take steps now in your current coding projects to reduce conversion effort later.

The first step is to understand what changes OpenStep brings and why these changes were made.

DESIGN GOALS FOR OPENSTEP

The OpenStep specification replaces the NEXTSTEP Release 3 Application Kit™, the Common classes and functions, and the DPS client library. OpenStep has many improvements over NEXTSTEP Release 3 that are based on these design goals:

· **Portability** OpenStep contains many new classes that provide a layer of operating system and hardware independence. These new classes provide access to operating system services such as threads, timers, interprocess and network communication, and the current date and time. Another new class, NSString, insulates your code from the different character encodings used in different environments. All of the classes that provide operating system independence are defined in the Foundation Kit™.

· **Improved support for Distributed Objects** OpenStep includes the classes that distribute objects across processes and across machines. Using Distributed Objects improves portability because it allows you to share information between processes without using operating system calls. OpenStep also provides a protocol that allows you to use the same method to encode objects for distribution that you use to encode objects for archiving.

To allow you to distribute data more easily, many of the structures provided in NEXTSTEP Release 3 are now objects. For example, events, exceptions, colors, screens, and DPSContexts are now objects. Turning these structures into objects has other advantages: It allows you to use Objective C message syntax to archive, allocate, deallocate, and access this data. It also provides better encapsulation and future extensibility.

· **Allow creation of internationalized applications** The Foundation Kit provides a new object called NSString. NSString allows you to perform character manipulation on strings without requiring that you know which character encoding is being used. Using NSStrings, you can write truly internationalized code, code that will work with any writing system supported by the Unicode standard. The Text object and many other Application Kit objects have been modified to use NSString objects in place of C strings. OpenStep also supports complete localization of an application's user interface, the messages displayed to the user, and the presentation of dates
and times.

· **Improved memory management** To protect you from accessing freed objects and from never freeing your objects, OpenStep introduces a new scheme for automatically deallocating objects when they are no longer needed. This scheme uses reference counting to ensure that an object
is not deallocated while it is still being used. When you create an object, you can mark it for later release. The object is then added to a pool of objects whose reference counts are

decremented at the top of the event loop. When an object's reference count reaches 0, the object is deallocated.

In this new scheme, an object is always deallocated by the object that created it, meaning that you don't have to worry about deallocating an object that you receive from an Application Kit object. Because the Distributed Objects system is now part of OpenStep, it uses this new memory management scheme and follows this same rule. Thus, the OpenStep API works the same remotely as it does locally.

- **Cleaner, simpler API** Even in classes that have no conceptual changes, some improvements have been made to the API. To begin with, arguments and return values are statically typed to allow better compile-time type checking. Also methods that used to return **self** by convention now return **void**. All classes, functions, types, and constants have the prefix **NS** to better distinguish them from your classes and functions. Method names have been expanded so that they are more easily understood.

In addition to these API changes, which apply to virtually all classes, some of the classes with more complex APIs have been simplified. One example of this is PopUpList. PopUpList defines pop-up and pull-down lists. These lists are activated by a trigger button, and the individual items in the list are stored in a Matrix. When you create what looks like a pop-up list in Interface Builder, you are really creating the trigger button. To access the PopUpList, you must send **target** to the trigger button that you created in Interface Builder, and to access the individual items in the list, you must first retrieve the Matrix from the PopUpList with the **itemList** method, then access the items through the Matrix.

In OpenStep, this has all been simplified. A single object, NSPopUpButton, encapsulates the PopUpList, the Button, and the Matrix. The object created by Interface Builder is the NSPopUpButton, so you don't need to bother retrieving the trigger button's target anymore.

You can retrieve items in the list by sending messages directly to NSPopUpButton.

These changes are significant enough that you will need to convert existing applications before you can run them under OpenStep. NeXT will provide conversion tools to make converting applications easier; the conversion process is described later in this article.

IMPORTANT OPENSTEP DIFFERENCES

Many of the differences between OpenStep and NEXTSTEP Release 3 are fairly straightforward: A method or class in Release 3 is replaced by a new method or class, one to one. Other changes are more complex. Some entire sections of functionality have changed. You will need to understand these changes if you intend to update your applications for OpenStep.

In particular, there are these major areas of difference:

- Strings
- Changes to the DPS client library
- Updating the display of Views and Windows
- New objects: NSException, NSColor, NSEvent, NSScreen, and NSUserDefaults
- Notifications and delegates
- API conventions
- Archiving

The changes to allocation and deallocation and to strings are part of Foundation Kit, which was first introduced as part of the Enterprise Objects™Framework. You can find out more about them in the Foundation Kit documentation, as well as in ªSneak Preview: The New

Foundation Kitº (*NXApp* Summer 1994, pp. 3±28).

The following sections describe the other important differences between OpenStep and previous versions of NEXTSTEP.

Changes to the DPS Client Library

In OpenStep, extensions to the DPS client library use objects wherever possible. You now use objects to create and manage DPS contexts, to manage the application event loop, and to create and manage timed entries.

The new NSDPSContext class defines DPS context objects. This class provides methods that perform any operation you typically perform on a context. If you are more familiar with the C function interface, you can use the method **DPSContext** to retrieve the object's DPSContext record. You can then operate on this context record using any of the functions or single operator functions defined in the DPS client library. The object and the record are always synchronized with each other; if you change one, the other is updated as well.

To manage the application event loop, you now use an NSRunLoop object instead of DPS functions. The NSRunLoop class defines objects that manage input sources. You generally don't need to create the run loop explicitly; one is created for each thread of your application.

Timed entries in OpenStep are handled with NSTimer objects. The NSTimer class defines timer objects that work with NSRunLoop objects. When you convert your application, calls to the **DPSAddTimedEntry()** and **DPSRemoveTimedEntry()** functions will change as shown in the following example.

Old code

```
void myHandler(DPSTimedEntry teNumber, double now, void *who)
{
```

```
    [(id)who tick];
}
...
- setRefreshSpeed:(double)theSpeed
{
    refreshSpeed = theSpeed;
    if (mvFlags.running) {
        DPSRemoveTimedEntry(timedEntry);
        timedEntry = DPSAddTimedEntry(refreshSpeed, myHandler,
        self, NX_RUNMODALTHRESHOLD);
    }
    return self;
}
...
- tick
{
    ...
}
```

New code

```
- setRefreshSpeed:(double)theSpeed
{
    refreshSpeed = theSpeed;
    if (mvFlags.running) {
        [timedEntry invalidate]; [timedEntry release];
        timedEntry = [[NSTimer scheduledTimerWithTimeInterval:
            (NSTimeInterval)refreshSpeed
            target:self selector:@selector(tick:)
            userInfo:nil repeats:YES] retain];
    }
    return self;
}
...
- (void)tick:(NSTimer *)theTimer
```

```
{
    . . .
}
```

The **scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:** method creates a timer object that repeatedly schedules itself to fire after **refreshSpeed** seconds. It also registers the timer with the currently active NSRunLoop in the default mode. When the NSTimer created in this example fires, it sends the message **tick:** with itself as an argument to the object that
created the NSTimer. The **userInfo** argument is a place to specify any other information that the method might need. The **myHandler()** function is no longer needed because it existed only to invoke the **tick** method.

Automatically Updating the Display

The Application Kit as it is today has a number of mechanisms to update the display of Views and Windows, and many different classes provide various degrees of control over them. OpenStep condenses all of these mechanisms into a single mechanism that 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.

In OpenStep, when you make a change that affects a view's appearance, the following sequence of events occurs:

1. The view sets its **needsDisplay** flag.

2. After the current user event has completed, the window receives an **update** message.

3. **update** checks all of the views in the window's view hierarchy. If a view's **needsDisplay** flag is set, it is redrawn.

You can disable this automatic updating of the display with NSWindow's **setAutodisplay:** method. When this mechanism is disabled, the window still receives the **update** message after each event, but it does not redraw the views. 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.

The following table lists the Application, View, and Window methods that are obsolete because of this change. It also shows what OpenStep method replaces that method.

Obsolete method	**OpenStep replacement**
autoupdate (in Application)	isAutodisplay (in NSWindow)
setAutoupdate: (in Application)	setAutodisplay: (in NSWindow)
disableDisplay (in Window)	None
enableDisplay (in Window)	None
isDisplayEnabled (in Window)	None
invalidate:: (in View)	setNeedsDisplay:YES
isAutodisplay (in View)	None
setAutodisplay: (in View)	None (use display directly)
setsDisplayOnScroll: (in ClipView)	None
update (in View)	displayIfNeeded

OpenStep also simplifies the automatic enabling or disabling of menu cells. Currently when you want a menu cell to be updated automatically, you implement a method that determines if the menu cell should be enabled or disabled, and you send the name of that method to the MenuCell in a **setUpdateAction:forMenu:** message.

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

In a few instances, you may still need to control whether a cell is enabled or disabled yourself.
For example, suppose you have an object that implements a **copy** method, but it should be used only to copy TIFF images. If your object is the first responder, the Copy command should be enabled only if a TIFF image is selected. In this case, your object should implement the method **validateCell:** to test whether the selection is a TIFF image and to enable or disable the Copy command accordingly. Before NSMenu enables the cell, it looks to see if the object that implements the action method also implements **validateCell:**. If so, it invokes that method to determine if it should enable the cell. **validateCell:** is defined in the NSMenuActionResponder informal protocol.

Notifications and Delegates

The Foundation Kit introduces a notification system, which is a way for objects that don't know about each other to communicate. Every application that uses OpenStep has a notification
center (an instance of the class NSNotificationCenter). One object tells the notification center that a particular event has occurred, and the notification center broadcasts that event to all interested objects.

The notification system is similar to using delegates, but it has these notable differences:

· Any number of objects may receive the notification, not just the delegate object.

· An object may receive any message you like from the notification center, not just the predefined delegate methods.

· The object posting the notification does not even have to know the other object exists.

· Because there may be many objects receiving the same notification, none of the receiving

objects can pass a value back to the object that posted the notification.

In OpenStep, some Application Kit objects that previously used delegates use the notification system to inform delegates that an event occurred. If the Application Kit object must receive a value back from the delegate, it still sends a message directly to the delegate.

Objects with delegates are not the only objects that post notifications. Any object can post a notification. The rest of this section explains how you can use the notification system in your application and how the use of the notification system will affect your existing delegate methods.

Using the notification system

If you want an object to receive a notification about a particular event, you register that object with the notification center. To register the object, have it send this message to an NSNotificationCenter (typically, the application's default notification center):

± (void)**addObserver:**(id)*recipient* **selector:**(SEL)*message*
name:(NSString *)*notification* **object:**(id)*anObject*

Once you send this message, whenever the center receives a notification *notification* from object *anObject*, it sends *recipient* the message *message*. You can specify **nil** for *anObject*, which means that any time the notification center receives *notification* (from any object), it should notify *recipient*.

In the example below, the sending object will receive the **windowMoved:** message whenever the object **importantWindow** posts NSWindowDidMoveNotification to the application's default notification center.

```
[[NSNotificationCenter defaultCenter]
   addObserver:self
   selector:@selector(windowMoved:)
```

```
    name:NSWindowDidMoveNotification
    object:importantWindow];
```

The message that the notification center sends to your object (**windowMoved:** in this example) must take an NSNotification object as an argument. An NSNotification object is the only type
of object you can post to a notification center. The NSNotification object contains a name (which objects use to identify the notification when they post it and when they register to receive it),
the object that posted it, and sometimes extra information stored in a userInfo dictionary. Your method can use the **[notification object]** message to find out which object posted the notification and can retrieve other necessary information using **[notification userInfo]** to access the
userInfo dictionary.

Changes to delegates

Although Application Kit objects that send messages to delegates will do so by posting notifications to the notification center, you won't have to perform any extra steps to have your delegate work the same way it always did. Delegates are registered to receive notifications automatically. Inside the delegate method itself, there are some minor changes. This section describes those changes.

A delegate method now receives an NSNotification object as an argument where it used to receive the object that sent the message (that is the object for which it is a delegate). To retrieve the object that used to be the sender of your delegate method, use the NSNotification method **object**, as shown in the following example.

Old code

```
- windowDidUpdate:sender
{
```

```
   if ([sender isMainWindow])
      [view updateLinksPanel];
   return self;
}
```

New code

```
- windowDidUpdate:(NSNotification *)notification
{
   NSWindow *theWindow = [notification object];
         /* the "sender" */

   if ([theWindow isMainWindow])
      [view updateLinksPanel];
   return self;
}
```

Matrix and TextField delegates

Previously, Matrix and TextField objects sometimes had a text delegate that responded to the actions that the field editor took. (The field editor is the NSText object used to draw and edit text in a matrix or text field.) These two objects now have their own delegates. Any field editor delegates defined in your application are converted to NSMatrix or NSTextField delegates. Instead of responding to NSText delegate methods, these new delegates implement the
methods shown in the following table.

Old delegate method	Replacement
textDidChange:	controlTextDidBeginEditing:
textDidEnd:endChar:	controlTextDidEndEditing:
textDidGetKeys:isEmpty:	controlTextDidChange:

Previously, the field editor delegate received a Text object as an argument to its delegate methods, and it was very hard to find out which control sent the message. Now, the field

editor delegate receives an NSNotification object. The sending NSMatrix or NSTextField object can be retrieved with the message **[notification object]**. You can retrieve the NSText object from the NSNotification's userInfo dictionary. The following example shows how to retrieve the NSText object from the NSNotification.

Old code

```
± textDidChange:sender
{
   Text *fieldEditor = sender;
   ...
}
```

New code

```
± controlTextDidBeginEditing:(NSNotification *)notification
{
   NSText *fieldEditor = [[notification userInfo]
   objectForKey:@"NSFieldEditor"];
   ...
}
```

New Objects

Many of the structures that you're used to dealing with are objects in OpenStep. For example, exceptions, colors, events, screens, and user defaults all now use objects rather than structures. As we mentioned earlier, using objects for these items allows you to distribute the information they contain, to archive that information, and to use a unified way of allocating and deallocating storage.

Exceptions

OpenStep uses NSException objects during exception handling. The NSException object

describes the exceptional condition. You write exception handlers in a similar fashion as you you do now, but you raise an exception differently.

The NSException class introduces these changes to the way you now define an exception:

· NSException uses names rather than numbers to identify exceptions.

· You always associate an error message with an NSException.

· You provide any necessary application-specific data through an NSDictionary, which is a new Foundation Kit class that stores key-value pairs.

· You use the variable **localException** where you used to use **NXLocalHandler**.

To create an NSException object, send the **exceptionWithName:reason:userInfo:** message to the NSException class object. The first argument to this method is an NSString containing the name of the exception. The second argument is an NSString containing an error message that states the reason why the exception occurred. The third argument takes an NSDictionary object in which you supply any necessary information to the exception. The following example shows how you would convert an **NX_RAISE()** macro call to code that creates an NSException.

Old code

```
int returnValue;
...
returnValue = aFunction();
if (returnValue)
    NX_RAISE(AFUNCTON_ERROR, &returnValue, NULL);
```

New code

```
int returnValue;
...
```

```
returnValue = aFunction();
if (returnValue) {
    NSException *theException = [[NSException
        exceptionWithName:@"aFunctionException"
        reason:@"Error during aFunction"
        userInfo:[NSDictionary dictionaryWithObjectsAndKeys:
         [NSNumber numberWithInt:returnValue], @"Return Value", nil]
         raise];
}
```

Colors

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. Application Kit objects now use NSColor objects to define their color values. In addition, Application Kit objects no longer take a separate gray value and color value. Currently, you have 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.

All NSColor objects have an alpha components, but you can use the **setIgnoresAlpha:** method to enable or disable the use of the alpha components.

Events

NSEvent objects represent an event from the application event queue. You can retrieve the same information from an NSEvent object that you can now retrieve from an NXEvent structure. NSEvents are constant objects, so the information in an NSEvent object won't change out from under you.

Screens

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 object 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.

User defaults

The OpenStep Foundation Kit provides a new user defaults system managed by an object of class NSUserDefaults. In this new system, defaults are stored in separate domains. Within each domain, the defaults are stored in NSMutableDictionary objects. Basically, you perform two functions with NSUserDefaults, modifying the defaults and retrieving the defaults:

- To add a default or to change a default already in the system, you use a method such as **setObject:forKey:**. NSUserDefaults provides several methods to add or change a value. You choose a method based on the type of value you want to store (array, integer, and so on). Saving the default values is automatic.

- To retrieve the value for a default, use a method such as **objectForKey:**. Again, NSUserDefaults provides several methods to perform this function, and you choose which one to use based on the type of value you want to retrieve.

When you request a default value, the NSUserDefaults object searches the domains in its search list in the order defined by the search list. When it finds the first occurrence, it stops the search. You can add to, remove from, or rearrange the order of the domains in the search list. The default search list is:

1. The argument domain, which contains defaults parsed from the application's command-line arguments
2. The application's domain

3 The domains for each of the user's preferred languages

4 The global domain, which contains defaults seen by all applications

5 The registration domain, which contains temporary defaults whose values can be set by the application to ensure that searches will always be successful

New API Conventions

OpenStep follows these new API conventions:

- All instance variables are private.
- Methods don't return **self** without a good reason.
- Method arguments and most method return values are statically typed.

All instance variables are private

Allowing direct access to instance variables violates encapsulation, so all instance variables are private in OpenStep. Hiding instance variables not only protects them better, but it makes it easier for objects to change without breaking subclasses.

Because all instance variables are private, if you subclass an Application Kit class, you must now use accessor methods to access the superclass's instance variables. Methods have been added to query and set all instance variables where they did not previously exist.

Old code

```
originalWidth = bounds.size.width;
```

New code

```
originalWidth = [self bounds].size.width;
```

Methods return void by default

Currently, methods return **self** by convention. Some methods return **self** to indicate success and **nil** to indicate failure. Returning **self** to indicate a Boolean value or returning **self** without any associated meaning made the API more confusing. In OpenStep, when a method has no real value to return, its return type is **void**. Where a method returns **self** or **nil**, its OpenStep counterpart returns **BOOL**.

Arguments are statically typed

To make the API more descriptive and explicit, all method arguments are now statically typed. Static typing provides better compile-time type checking, plus it makes it easier for you to learn how to use a method.

Old code

```
± (int)browser:sender numberOfRowsInColumn:(int)column ...
```

New code

```
± (int)browser:(NSBrowser *)sender numberOfRowsInColumn:(int)column ...
```

New Archiving Scheme

The new root object, NSObject, introduces major changes to the archiving scheme. First, archives are written to NSData objects instead of to NXTypedStreams. NSData defines objects that are generic data buffers. A related class, NSMutableData, contains data that you can modify.

Second, there are two new objects in the Foundation Kit, NSArchiver and NSUnarchiver, that

archive your application's objects and remove your objects from the archive, respectively. Both NSArchiver and NSUnarchiver are subclasses of the same abstract superclass, NSCoder.

NSCoders are objects that know how to represent an object in a different format: a format for archiving to a file, a format for shipping an object to another process, or any other format you might identify. In OpenStep, both the archiving system and the Distributed Objects system use NSCoders, so you no longer have to write two sets of methods if you want to both archive and distribute copies of your object. However, writing one set of methods to do both operations also means that you need to pay attention to a few more details in the single set of methods you do write. This is described more later in this section.

Archiving and unarchiving objects

When you archive a set of objects in OpenStep, the following sequence of events occurs:

1. You create an instance of the NSArchiver class.
2. You send either **encodeRootObject:** or **archiveRootObject:toFile:** to the NSArchiver.
3. The NSArchiver sends the root object an **encodeWithCoder:self** message.
4. Each object in the object graph is eventually sent an **encodeWithCoder:** message.

encodeWithCoder: replaces the **write:** method. In the body of each **encodeWithCoder:** method, the NSArchiver is called on to archive that object's instance variables. It does so by writing them to an NSMutableData object.

When you unarchive a set of objects, a similar sequence of events occurs:

1. You create an instance of the NSUnarchiver class, usually configuring it with the NSData object to which you wrote the archive.

2 You send either **unarchiveObjectWithFile:** or **decodeObject** to the NSUnarchiver.

3 The NSUnarchiver sends the first object in the archive the message **initWithCoder:self**.

4 Each object in the graph is eventually sent the **initWithCoder:** message. (This method replaces the **read:** method.)

Converting the read: and write: methods

In OpenStep, **read:** and **write:** are replaced by **initWithCoder:** and **encodeWithCoder:**. The method **initWithCoder:** reads values from an NSUnarchiver object and assigns them to the object's instance variables. Conversely, **encodeWithCoder:** writes the object's instance variables to an NSArchiver object. These methods are defined in the NSCoding protocol.

If you have an object that you want to archive, it must conform to the NSCoding protocol. In **initWithCoder:** and **encodeWithCoder:**, you must follow many of the same rules you followed in the **read:** and **write:** methods. In particular, you must call parallel methods in **initWithCoder:** and **encodeWithCoder:**, just like you must call parallel functions in **read:** and **write:**, and you must retrieve data from the archive in the same order in which you placed it in the archive.

For example, if your **encodeWithCoder:** method looks like the one shown below, your **initWithCoder:** method must look like the one shown below as well.

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [super encodeWithCoder:aCoder];
    [aCoder encodeObject:authorID];
    [aCoder encodeValuesOfObjCTypes:"@", &phone];
    [aCoder encodeValuesOfObjCTypes:"i", &contract];
}
```

```
- initWithCoder:(NSCoder *)aDecoder
{
    [super initWithCoder:aDecoder];
    authorID = [[aDecoder decodeObject] retain];
    [aDecoder decodeValuesOfObjCTypes:"@", &phone];
    [aDecoder decodeValuesOfObjCTypes:"i", &contract];
    return self;
}
```

This example shows **encodeObject:** as the basic method you use to archive an object. However, if you want to encode an object for both archiving and distribution, you instead use **encodeBycopyObject:**. When the object is distributed, it is distributed as a proxy. (Proxies inherit from the NSProxy class instead of NSObject.) **encodeBycopyObject:** encodes the object in such a way that when it is decoded, a copy of the object is returned rather than the proxy.

In NSArchiver, **encodeBycopyObject:** simply calls **encodeObject:** and returns. This way, if **encodeWithCoder:** is passed an NSArchiver as the code, it archives the object in the usual way.

CONVERTING TO OPENSTEP

Because of all of the differences described above, you must convert any Release 3 application before you can run it on NEXTSTEP 4.0 or another OpenStep-compliant system. Fortunately, since NEXTSTEP 4.0 conforms to the OpenStep specification, converting your code to NEXTSTEP 4.0 is a good way to turn your application into an OpenStep application. There are a few operations outside of the specification's scope that your application may have to perform, such as accessing ports. However, converting will make your code as portable as possible.

As mentioned earlier, NeXT will provide tools that make converting applications easier. In a few places, the conversion tools may change your code to use constructs defined in NEXTSTEP 4.0 but not in OpenStep. These places will be clearly documented in the conversion guide that accompanies Release 4.0.

For now, we can give you a quick preview of the conversion process. You run it in six main stages, with an optional seventh stage at the end:

· Stage 1 performs the most basic level of the conversion on your code. It converts the names of Application Kit objects and many methods. It also converts the root object: All of your objects now inherit from the new root object, NSObject. It updates the object allocation and deallocation mechanisms, all of the List, C string, and NXRect arguments used in the Application Kit, and the exception handling mechanism.

· Stage 2 converts streams, text, and images. Streams are no longer used in the OpenStep API, so during stage 2, all uses of streams in your code are converted to various OpenStep objects, usually NSData objects. The most notable use of streams is in archiving, so the new archiving scheme is introduced in this stage.

· Stage 3 converts code that accesses the Window Server. As you would expect, there have been many changes to this portion of NEXTSTEP to make it operating system±independent. Classes from the Foundation Kit operating system±independence layer, NSEvent, NSTimer, and NSRunLoop, are introduced at this stage. Another new class, NSDPSContext, defines objects representing the PostScript execution context. Also, colors and fonts are converted to their new implementations at this stage.

· Stage 4 converts the parts of your application that display on the screenÐnamely, views and windowsÐand printing. The API for views and windows has been simplified in many areas, making many of the existing View and Window methods obsolete. Most of the

changes in this area involve removing View and Window methods that are no longer necessary. In printing, many of the duties of the shared PrintInfo object have been off-loaded to a new Application Kit object, NSPrintOperation. The changes to printing will be mostly transparent to you unless you create and control your own printing processes.

· Stage 5 converts several other parts of your application. First, it converts delegate objects, because some Application Kit objects with delegates use OpenStep's new notification system, described earlier. It also converts the Application objectÐmany of the duties of the Application object have been off-loaded to other, more appropriate, objects in the Application Kit. As a result, you need to send messages to **NXApp** less.

· Moreover, this stage updates Workspace requests. The NSWorkspace class replaces the NSWorkspaceRequestProtocol protocol. In addition, applications now store and access their user defaults through the NSUserDefaults object instead of Application Kit functions. And finally, this stages converts Controls and Cells: These objects have not changed much, but there have been a few significant improvements of which you may want to take advantage.

· Stage 6 performs an API cleanup. All OpenStep API follow certain conventions that the NEXTSTEP kits did not necessarily follow. Use of these conventions requires some changes to be made to your code. In addition, it makes changes to modal panels. The use of Application Kit modal panels (such as the Open and Save panels) is slightly different. Modal panels are handed out differently, and modal panels that use file names have been simplified so that they always use the absolute path.

After you've performed the six required conversion stages, you can run an optional conversion stage. The optional conversions change occurrences of the Common classes in your code. The Common classes are obsolete in OpenStep, replaced by classes in the Foundation Kit. The required conversions change all of the places where the Application Kit gave you an instance of a Common class. The optional conversions change places where

you create an instance of a Common class yourself. These conversions are optional because the Common classes are still supported in this release of NEXTSTEP. However, if you want to make your code as portable as possible, you should run them.

PREPARING FOR OPENSTEP IN YOUR CURRENT WORK

While the conversion tools will simplify the process of moving applications to OpenStep, you might want to take steps now to save effort later. In the applications you're currently writing, you can begin to adopt some of the OpenStep ideas. This will make it easier to use the conversion tools in the future.

· **Use Foundation Kit classes.** Because OpenStep is based on Foundation Kit, using Foundation Kit classes now is certainly good preparation for future work. Additionally, the Foundation Kit classes and memory management facilities are much nicer than what we've had before in their place.

The most important Foundation Kit objects to start using early are NSArray, NSDictionary, and NSString. The old classes List, HashTable, and Storage will soon be obsolete, so you can save yourself trouble later by using NSArray and NSDictionary in their place. And using NSString now will ease the process of melding your app to the OpenStep Application Kit, which uses NSString in all its APIs in place of **(char *)**. Using NSString now will also allow your code to be fully international in Release 4.0.

· **Use Distributed Objects.** Listener and Speaker are not part of OpenStep, because Distributed Objects (DO) provides a much more powerful mechanism and is operating system±independent. Converting uses of DO to OpenStep will be much easier than converting uses of Listener/ Speaker. There is no reason not to use DO now.

· **Use ANSI routines.** ANSI C library routines will be provided with any version of OpenStep, whereas UNIX®-specific routines may not be available on all platforms supporting OpenStep.

· **Avoid direct access to instance variables.** For better encapsulation and extensibility, instance variables in OpenStep aren't part of the API. Using methods to access the information from subclasses' code now eliminates later conversion work.

· **Don't rely on returning self.** As mentioned earlier, methods in OpenStep return **void** instead of **self** by default. Avoid writing code that nests a set of message expressions and relies on the current convention of returning **self**.

· **Use displayIfNeeded.** For updating a subset of controls in a window, the **displayIfNeeded** mechanism is a close approximation to the display updating mechanism OpenStep will use.

(In contrast, the most dissimilar strategyÐand therefore the one that will be most difficult to convertÐis one in which each view that is updated is sent an explicit **display** message.)

· **Use NX functions.** Your code will be more portable if you use Application Kit objects and **NX** functions wherever possible instead of directly calling the PostScript operators that NeXT added to Display PostScript. For example, use the Cursor object instead of the **setcursor** PostScript operator.

Two exceptions are the **alpha** and **compositing** operatorsÐthey're included in OpenStep.

· **Avoid NXJournaler and SelectionCell.** These classes aren't in OpenStep. Journaling is too window-system dependent to be included in OpenStep, while SelectionCell was a trivial cell class provided in Release 1.0, and replaced in Release 2.0 by the complete NXBrowser class.

DON'T WAIT, ACT NOW!

With NEXTSTEP Release 4.0 you'll receive the tools and documentation you'll need to upgrade applications and take full advantage of new capabilities. But that doesn't mean you should wait until then to prepare. Start using Foundation Kit now, and be sure to avoid the soon-to-be-history methods and classes listed in this article. You'll be well on your way to having portable applications that you can run on NeXT's and other vendors' OpenStep implementations.

Jean Ostrem is a member of NeXT's Developer Publications group. You can reach her via e-mail at **Jean_Ostrem@next.com.**

Next Article NeXTanswer #1991 **Writing Device Drivers in an Object-Oriented World**
Table of contents

http://www.next.com/HotNews/Journal/OSJ/SpringContents95.html