Chapter 1

Global API and Style Changes

The conversion scripts described in this chapter perform API style changes. If you're converting in stages, you'll see a portion of these changes after stage 1 and the rest of them after stage 6.

Factory Methods Conversion

Stage 1

As stated in the introduction, OpenStep introduces a new scheme for automatically deallocating objects when they are no longer needed. This scheme, which is described in detail later (in the section ªNSObject Conversionº), changes the typical way an object is created.

Where you used to use **alloc** and **init** to allocate a new object, you typically now use a ***+classname*** method. These methods are called ***+classname*** methods because their names begin with the name of the class minus the NS prefix. They are also called factory methods. By convention, the ***+classname*** methods always perform allocation, initialization, and autorelease. (The **autorelease** method is described in ªNSObject Conversion.º It makes sure the object is deallocated when you no longer need it.) The ***+classname*** methods essentially replace **new** methods, which you used to perform both allocation and initialization.

For example, the Foundation Framework contains the NSNumber class, which defines a number object. You can allocate an NSNumber instance, initialize it to have a specific integer value, and release it at the top of

the event loop by sending NSNumber this one message:

New Code

```
NSNumber *intObject = [NSNumber numberWithInt:anInt];
```

Wherever possible, any **alloc** and **init** or **new** messages sent to an Application Kit class are converted to a ***+classname*** method so the object will be released at the top of the event loop without your having to perform any extra steps. However, most Application Kit objects don't define a ***+classname*** method because you typically want them to stick around for more than one event cycle. If you've used a **new** method to create an Application Kit object, this conversion replaces it with the appropriate **alloc** and **init** messages.

Two examples of Application Kit classes with ***+classname*** methods are NSOpenPanel and NSSavePanel. When you use **openPanel** or **savePanel** to request an open or a save panel, the class checks to see if a panel has previously been created. If so, the class returns the previously created panel, but first it removes any changes you made to the panel.

Because of this change to the open and save panels, you might have to complete the conversion manually if your application makes changes to these panels. Previously, your application would have to remove the changes it made to a modal panel before the panel was used again. Now, removing such changes is done for you, so that you always receive the default panel each time you use its ***+classname*** method. If you have code to return a modal panel to its default state, you may delete it. For more information, see the class specification for each of the modal panels in the *Application Kit Reference*.

General Naming Conversion

Stage 6

The general naming conversion performs the trivial name changes. In many cases, only the name has changed, but the class or method still operates in the same manner. For example, **conformsTo:** has changed to **conformsToProtocol:** so that its purpose is more easily understood. Most of the changes that this conversion makes require no intervention; however, some may require you to inspect an argument's type and make sure that its still valid. The conversion flags these changes with a warning or error message.

Validating and Filtering Files for Open and Save Panels

A few changes have been made to the way Open and Save panels handle filenames. The table below summarizes these changes.

Old Method Name	**New Method Name**	**Other Changes**
TableHeadRule.eps ¬		
filenames	filenames	Now returns absolute paths for each file. If you were forced to construct absolute paths from the return value, you may be able to remove some code.
TableRule.eps ¬		
panel:filterFile:inDirectory:	panel:shouldShowFilename:	The filename argument is now an absolute path.
TableRule.eps ¬		
panelValidateFilenames:	panel:isValidFilename:	Files are validated one at a time. The new method is invoked as many times as necessary to validate all selected files.
TableRule.eps ¬		

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 ªDefaults Conversionº in the chapter ªConverting the Common Classes.º

NX_ZONEMALLOC and NX_ZONEREALLOC

The Application Kit macros **NX_ZONEMALLOC** and **NX_ZONEREALLOC** are replaced by the Foundation Framework functions **NSZoneMalloc()** and **NSZoneRealloc()**. Sometimes, the conversion process is unable to convert your use of these macros. In these cases, it will print an error, and you must perform the conversion manually. The definitions of **NSZoneMalloc()** and **NSZoneRealloc()** are:

void€*NSZoneMalloc(NSZone€**zone*, unsigned€*size*);
void *NSZoneRealloc(NSZone€**zone*, void€**pointer*, unsigned€*size*);

Ivar Conversion

Stage 6

All instance variables are private in OpenStep. Allowing direct access to instance variables violates encapsulation. Hiding instance variables not only protects them better from inadvertent changes, but also 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. The ivar conversion makes this change for you. (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;
```

The conversion process assumes that you never set an inherited instance variable's value directly through an assignment statement, because this was not supported in Release 3.3. If you do, you may see the following in your code after this conversion is complete.

Old Code

```
bounds.size.width = 7.5;
```

Bad New Code

```
[self bounds].size.width = 7.5; /* Compile Time Error! */
```

To correct this, use the appropriate superclass method to set the instance variable's value, as shown here.

Good New Code

```
NSSize newSize;

newSize.height = [self bounds].size.height;
newSize.width = 7.5;
[self setBoundsSize:newSize];
```

NSName Conversion

Stage 1

The NSName conversion does two things:

SquareBullet.eps ¬Adds the prefix ªNSº to all keywords except method names. For example, View is now NSView. This prefix distinguishes these keywords from their old implementations and from keywords you create.

SquareBullet.eps ¬Capitalizes the main header files to distinguish them from the old implementations. For example **<appkit/appkit.h>** is converted to **<AppKit/AppKit.h>**.

These name changes are the most immediately visible differences after stage 1. In many cases, the name has changed, but the class operates in the same manner. However, this conversion catches many of the class, types, and functions that are obsolete. Some of these have no replacement in OpenStep. If the conversion finds an obsolete item in your code, it prints an error that tells you why the item is obsolete.

NXZone and Shallow Conversions

The NSName conversion changes all NXZone references to NSZone. NSZone is a structure defined in the Foundation Framework. If you're performing a shallow conversion, you'll still have instances of the Common classes in your code (such as List and Storage), and those classes still use the NXZone structure defined in **objc/zone.h**. The definitions of NSZone and NXZone are identical. You can safely cast the NSZone structure returned by the NSObject **zone** method, for example, to an NXZone structure.

NSObject Conversion

Stage 1

All objects now inherit from NSObject, the new root class that is part of the Foundation Framework. All of the classes included in OpenStep inherit from NSObject, and they all begin with the prefix ªNS.º So if the classes defined in your application now inherit from a class beginning with ªNS,º they ultimately inherit from NSObject.

NSObject replaces Object. It provides all of the mechanisms that Object did. Some of these mechanisms, for example object allocation and deallocation, have changed. However, the basis for object allocation and deallocation is still provided by the root class.

Object Allocation and Deallocation

NSObject introduces a new scheme for object allocation and deallocation. In converting your code to use this new scheme, the NSObject conversion replaces many invocations of the **alloc**, **init**, **new**, and **free** methods. From outward appearances, it may seem like not much has changed except for the names of the methods you use to allocate and deallocate objects. In reality, these methods operate in a very different way, and it is very important to understand how they work.

Note: For more information about object allocation and deallocation, see the introduction to the *Foundation Framework Reference*.

In the new scheme, each object keeps track of the number of objects that refer to it. This number is called the object's *reference count*. An object can't be deallocated unless its reference count is 0Ðthat is, any objects that referred to it are finished with it.

The methods in the following table increment an object's reference count.

Method Name TableHeadRule.eps ¬	Purpose
alloc TableRule.eps ¬	Allocates a new object.
copy TableRule.eps ¬	Creates a copy of an existing object.
mutableCopy TableRule.eps ¬	Creates a modifiable copy of an existing object.
retain TableRule.eps ¬	Increases the reference count to ensure that an object does not get deallocated.

The methods in the following table decrement the reference count.

Method Name TableHeadRule.eps ¬	Purpose
release TableRule.eps ¬	Decrements reference count. If count becomes 0, invokes **dealloc** to deallocate the object.
autorelease TableRule.eps ¬	Indirectly decrements the reference count by sending the object the **release** message at the top of the event loop.

Previously, you used **alloc** to create an object and when you were done with the object, you sent it the **free** message. In OpenStep, you can still create an object with **alloc**, but when you are done with it, you send it **release** or **autorelease**. As stated in the table, **release** decrements the reference count, and if the reference count becomes 0, it invokes the object's **dealloc** method (**dealloc** replaces **free**). **autorelease** adds the object to a pool of objects in the application called the *autorelease pool*. At the top of the event loop, each object in the

autorelease pool is sent the **release** message. Again, if the reference count becomes 0 as a result of the **release** message, the object is deallocated.

The **free** method is replaced by **dealloc**. You never invoke **dealloc** directly; it is always invoked indirectly through **release**. The conversion process replaces all **free** messages with **release** messages and replaces all overrides of the **free** method with the **dealloc** method.

Automatically Releasing Objects

You usually use **autorelease** instead of **release** because it ensures that the object will not be deallocated until the end of the current event. For example, you can send **autorelease** anywhere inside a method implementation and continue to use the object, because the object won't be released until the method has completed.

Just as you typically put the **alloc** and **init** methods in the same message to make sure that you never use an uninitialized object, you also typically **autorelease** the object on that same line to make sure you never forget to decrement its reference count. The following shows how the typical object creation statement has changed. The new code shown below creates an object that will automatically be deallocated at the top of the event loop.

Old Code

```
id myObject = [[Object alloc] init];
```

New Code

```
id myObject = [[[NSObject alloc] init] autorelease];
```

Making Sure Objects Are Not Automatically Deallocated

The ***+classname*** methods return an object with a reference count of 1, but the reference count automatically becomes 0 after the current event has completed because ***+classname*** invokes **autorelease**. If you are creating an object as an instance variable, you need to increment its reference count so that it is not deallocated until your object is deallocated.

If you create an instance variable with a ***+classname*** method, use the **retain** method to increment its reference count. In your object's **dealloc** method, you send these instance variables the **release** message so that their reference counts become 0 and they are immediately deallocated. The following example illustrates this change.

New Code

```
- init
{
...
     oneIVar = [[NSNumber numberWithInt:1] retain];
...
}

-(void)dealloc
{
...
    [oneIVar release];
...
}
```

Objects Created in Interface Builder

If you created an instance variable in Interface Builder, you don't have to do anything different from what you

used to do. Custom objects unarchived from a nib file have a reference count of 1. If you have a custom object that you use as an instance variable of another object, release that instance variable in the **dealloc** method of the other object. Views created in Interface Builder are retained and released automatically. Windows are not released until the user quits the application unless you specify otherwise.

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.

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.

NSObject Gotchas

After the NSObject conversion, watch out for the following:

Retain Cycles

In general, you retain all instance variables you create in code. Sometimes you have two objects with

instance variables that refer to each other. For example, consider the architecture for a text document shown in the figure. The Document object creates a Page object instance variable for each page in the document. Each Page object has an instance variable that keeps track of which document it's in. If the Document object retained the Page object and the Page object retained the Document object, neither object would ever be released. The Document's reference count can't become 0 until the Page object is released, and the Page object won't be released until the Document object is deallocated.

As a rule of thumb, if your application has a similar object hierarchy, the ªparentº object should retain its ªchildren,º but the children should not retain their parents. This follows the same pattern as the view hierarchy Ð superviews retain their subviews as they are added to the hierarchy and release them as they are removed.

RetainCycles.eps ¬

Autorelease Memory Leaks

For the **autorelease** method to release and eventually deallocate an object, you must have an autorelease pool in place. If you're converting an application that uses NSApplication, you automatically get an autorelease pool. If you have an NSApplication subclass that overrides **init** or **run** or if you are converting a program that doesn't use the Application Kit (for example, a UNIX command-line tool), you must create your own autorelease pool. The easiest way to do this is to allocate an NSAutoreleasePool object as the first statement of your **main()** function and release it as the last statement before **main()** exits. If you don't have an autorelease pool, you receive warning messages at run time when the program sends an **autorelease** message.

You may want autorelease pools in other parts of your program as well, for example, in loops that allocate a lot of objects. For more information see the NSAutoreleasePool class specification in the *Foundation Framework Reference*.

New Code

```
void main (int argc, char *argv[])
{
     NSAutoreleasePool *myPool = [[NSAutoreleasePool alloc] init];
```

```
    /* other declarations go here. */
    ...
    [myPool release];
    exit (0);
}
```

Retrieving the Class Name

The Object class defined the **name** method, which returned the class name of the object as a C string. This method is obsolete in OpenStep. The conversion process doesn't catch the **name** method for you. You must change it manually.

In most cases, the proper replacement for **name** is **class**, which returns the class object. The OpenStep API never requires you to know the name of the class object; instead, it uses the class object. For example, **isKindOfClass:** and **isMemberOfClass:** both take class objects as arguments rather than strings. However, if you do need the name of the class, you can use the method **NSStringFromClass()** to retrieve it from the class object returned by the **class** method.

Old Code

```
printf("The class name is: %s\n", [myObject name]);
```

New Code

```
printf("The class name is: %s\n",
    [NSStringFromClass([myObject class]) cString]);
```

forward::

The **forward::** method used to take a selector and a pointer to that selector's arguments as its arguments. This method is replaced by **forwardInvocation:**, which takes an NSInvocation object as its argument. If you implement the **forward::** method, the conversion process flags it with an error. You should replace **forward::** with **forwardInvocation:** and replace the selector and arguments list with an NSInvocation object. For more information about NSInvocation, see its class specification in the *Foundation Framework Reference*.

Static Typing Conversion

Stage 6

To catch more errors at compile time, almost all method arguments are now statically typed. Static typing also makes it easier for you to learn how to use a method. The static typing conversion makes this change for you.

Old Code

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

New Code

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

Void Conversion

Stage 6

Previously, methods that had no other information to return returned **self** by convention. Some methods returned **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. Now, when a method has no real value to return, its return type is **void**. Where a method returned **self** or **nil**, it now returns **BOOL**.

During this conversion, both Application Kit methods and methods that you wrote are converted to return **void**. (When you set up your project to be converted, the conversion process scanned your code for methods that always returned **self** and built a script to convert them to **void**. For more information, see the on-line release note *Converting Your Code to OpenStep* in **/NextLibrary/Documentation/NextDev/ReleaseNotes**.) When a method is converted to return **void**, its definition is changed, and the **return self** statement at the end of its implementation is removed. In addition, because the return type is now **void**, the conversion process changes invocations of the method in the following places.

Return Statements

Old Code

```
return [aWindow makeKeyAndOrderFront:self];
```

New Code

```
[aWindow makeKeyAndOrderFront:self];
return aWindow;
```

Assignment Statements

Old Code

```
keyWindow = [aWindow makeKeyAndOrderFront:self];
```

New Code

```
keyWindow = aWindow;
[keyWindow makeKeyAndOrderFront:self];
```

Nested Messages

Old Code

```
[[aWindow reenableFlushWindow] flushWindow];        /* easy case */
[[[aView window] reenableFlushWindow] flushWindow]; /* hard case */
```

New Code

```
[aWindow enableFlushWindow]; /* easy case */
[aWindow flushWindow];

id newVar = [aView window]; /* hard case */
[newVar enableFlushWindow];
[newVar flushWindow];
```

Message Arguments

Old Code

```
[NXApp runModalFor:[aWindow makeKeyAndOrderFront:self]];
```

New Code

```
[aWindow makeKeyAndOrderFront:self];
[NSApp runModalForWindow:aWindow];
```