Chapter 2

Converting the Common Classes

OpenStep introduces a new kit, the Foundation Framework, which replaces the Common classes (such as Object, List, and HashTable) and provides operating-system independence. This Foundation Framework is a superset of the Foundation Framework in NEXTSTEP Release 3.3. The OpenStep Application Kit uses classes from the new Foundation Framework instead of the Common classes it once used. That is, instance variables and method return values that were instances of a Common class are now instances of a Foundation class. For example, wherever an Application Kit class once returned a List object, it now returns an NSArray object. This chapter describes the Foundation classes and those places where they are introduced into your code.

Foundation Classes

Like the Common classes, the Foundation Framework provides classes for collections, storage, and data types. However, the Foundation Framework moves beyond the Common classes to provide an operating-system independence layer and to support distributed objects. In addition, the classes that provide basic support for the Objective-C language are now in the Foundation Framework. The following table lists these classes by function.

Type of Objects	Classes

TableHeadRule.eps ¬

Objective-C language support	NSObject, NSBundle, NSException, NSAssertionHandler, NSInvocation, NSMethodSignature, NSAutoreleasePool
TableRule.eps ¬	
Basic data types	NSData, NSMutableData, NSNumber, NSValue
TableRule.eps ¬	
String data types	NSString, NSMutableString, NSScanner, NSCharacterSet, NSMutableCharacterSet
TableRule.eps ¬	
Collections	NSArray, NSMutableArray, NSSet, NSMutableSet, NSCountedSet, NSDictionary, NSMutableDictionary, NSEnumerator
TableRule.eps ¬	
Object persistence	NSArchiver, NSUnarchiver, NSCoder, NSSerializer, NSDeserializer
TableRule.eps ¬	
Date and time	NSDate, NSCalendarDate, NSTimeZone, NSTimeZoneDetail
TableRule.eps ¬	
Interobject communication	NSNotification, NSNotificationCenter, NSNotificationQueue
TableRule.eps ¬	
Operating-system independence	NSThread, NSLock, NSConditionLock, NSRecursiveLock, NSRunLoop, NSTimer, NSProcessInfo, NSUserDefaults
TableRule.eps ¬	
Distributed objects system	NSProxy, NSDistantObject, NSConnection
TableRule.eps ¬	

Class Clusters

Some of the classes listed above (for example, NSNumber) are actually class clusters. Class clusters group several private, concrete subclasses under a public, abstract superclass. For example, NSNumber is an abstract superclass. It has one subclass for each of the basic C types that store numbers (**int**, **unsigned int**, **long**, **short**, **float**, **double**, **long double**, and so on). When you create an instance of NSNumber, you actually receive an instance of one of these subclasses.

Class clusters simplify the interface. Having one class for numbers is much easier to manage than it would be if all of NSNumber's subclasses were public (NSInt, NSLongInt, NSShortInt, etc.). You use an instance of a class cluster the same way you use any other object. The only reason you need to know about class clusters is that there are some differences in subclassing a class cluster. If you don't want to subclass a class cluster, and in all likelihood you won't need to, then you don't have to be concerned about this new concept.

Some class clusters have more than one public superclass. For example, NSArray is a class cluster with two public superclasses: NSArray and NSMutableArray. An NSArray object cannot be modified, but an NSMutableArray object can. So if you want to create an array that can't be modified or won't be modified, you instantiate NSArray. If you want to modify the array, you instantiate NSMutableArray.

See the introduction to the *Foundation Framework Reference* for more information on this topic. Within the *Foundation Framework Reference*, any public superclass of a class cluster is documented as a class cluster. That is, the NSArray class specification is titled ªNSArray Class Cluster.º

Archiver Conversion

Stage 2

The new root object, NSObject, introduces major changes to the archiving scheme. First, NXTypedStreams

are obsolete in OpenStep, so archives are written to NSData objects instead. NSData is a new class cluster in the Foundation Framework. It defines objects that are generic data buffers. It has two public superclasses, NSData, which contains unmodifiable data, and NSMutableData, which contains modifiable data.

Second, two new objects in the Foundation Framework, NSArchiver and NSUnarchiver, 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 in more detail 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 upon to archive that object's instance variables. It does so by writing them to an NSMutableData object. The following example shows how you previously archived all objects in an object graph and how you perform the same task using OpenStep API.

Old Code

```
- (BOOL) archiveThisObject:(Object *)object
{
     NXTypedStream *stream;
     char archivePath[MAXPATHLEN+1];
     ...
     stream = NXOpenTypedStreamForFile(archivePath, NX_WRITEONLY);
     NXWriteRootObject(stream, object);
     NXCloseTypedStream(stream);
     return YES;
}
```

New Code

```
- (BOOL) archiveThisObject:(NSObject *)object
{
     BOOL b;
     NSString *archivePath;
     ...
     b = [NSArchiver archiveRootObject:object toFile:archivePath];
     return b;
}
```

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 or the file 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.)

The following example shows how you might have previously unarchived all objects in an object graph and how you perform the same task using OpenStep API.

Old Code

```
-(Object *)unarchiveObject
{
     NXTypedStream *stream;
     Object *object;
     char archivePath[MAXPATHLEN+1];
     ...
     stream = NXOpenTypedStreamForFile(archivePath, NX_READONLY);
    object = NXReadObject(stream);
    NXCloseTypedStream(stream);
     return object;
}
```

New Code

```
- (NSObject *) unarchiveObject
{
     NSObject *object;
     NSString *archivePath;
     ...
    object = [NSUnarchiver unarchiveObjectWithFile:archivePath];
    return object;
```

```
}
```

The conversion process converts as much of the archiving code as possible. However, you will need to perform some of the conversion yourself. The following table lists the changes you may need to make within the code that starts the archiving and unarchiving processes. You may also need to make some changes in the implementation of **encodeWithCoder:** and **initWithCoder:**. Those changes are described in the section ªConverting the read: and write: Methodsº in this chapter.

Obsolete Item	**Replacement**
TableHeadRule.eps ¬	
NXTypedStream variables TableRule.eps ¬	Change to NSMutableData or NSData.
NXOpenTypedStreamForFile(NX_WRITEONLY) TableRule.eps ¬	Change to [NSArchiver archiveRootObject:toFile:].
NXOpenTypedStreamForFile(NX_READONLY) TableRule.eps ¬	Change to [NSUnarchiver unarchiveObjectWithFile:].
NXFlushTypedStream() TableRule.eps ¬	None necessary. Remove this function.
NXFreeObjectBuffer() TableRule.eps ¬	None necessary. Remove this function.
NXReadObjectFromBuffer() TableRule.eps ¬	Use NSUnarchiver object as described above.
NXReadObjectFromBufferWithZone() TableRule.eps ¬	Use NSUnarchiver object as described above.
NXWriteObjectFromBuffer() TableRule.eps ¬	Use NSArchiver object as described above.

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.

Note: 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:

SquareBullet.eps ¬ **encodeWithCoder:** is invoked twice, so it must not have side effects that cannot happen twice. Also, you need to be sure that the object you write is the exact same object each time.

SquareBullet.eps ¬ **initWithCoder:** and **encodeWithCoder:** must invoke parallel methods, just as **read:** and **write:** must call parallel functions. So if you use **encodeValuesOfObjCTypes:"@"** to archive an object, you must use **decodeValuesOfObjCTypes:"@"** to unarchive it, not **decodeObject**.

SquareBullet.eps ¬ 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.

New Code

```
- (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;
}
```

The conversion process automatically converts your **read:** and **write:** methods to **encodeWithCoder:** and **initWithCoder:**. It also changes the function calls inside these methods to the appropriate NSCoder messages. When the conversion process is complete, you must look at the implementation of these methods and verify that they are correct. The following sections help you determine if the implementations of **encodeWithCoder:** and **initWithCoder:** are correct.

decodeObject and decodeValuesOfObjCTypes:

For backward compatibility, when **decodeValuesOfObjCTypes:** is used to unarchive an object, it returns an object that is already retained. **decodeObject** does not retain the object it unarchives. So if you use **decodeObject** to unarchive an instance variable, you must send the object the **retain** message.

Because it is unusual for a method to return a retained object as **decodeValuesOfObjCTypes:** does, you may find it easier to always use **decodeObject** to unarchive instance variables. This way, you do not have to remember that there is an implicit **retain** message in the method you have used. You can think of this as a rule that your **init** methods must always explicitly retain instance variables. Because **initWithCoder:** is an **init** method, it must follow this rule.

Remember that you must use the analogous method to archive the object. If you use **decodeObject** to

unarchive an instance variable, you must use **encodeObject:** to archive it.

Archiving Objects You Don't Retain

Sometimes two objects have instance variables that refer to each other. This often occurs when there is a hierarchical order to objects in your application: Each object keeps track of a ªparentº object and one or more ªchildº objects. In general, parent objects retain their children, but to ensure that all objects will eventually be deallocated, children don't retain their parents. Just as you want to make sure the parent object isn't retained too many times, you also should make sure the parent object isn't archived too often.

The general rule for archiving in such a situation is this: Use **encodeObject:** to archive all objects that your object retains (its children in this example), and use **encodeConditionalObject:** to archive objects you do not retain (the parent in this example). **encodeConditionalObject:** encodes an object only if another object archives it unconditionally. Otherwise, it makes the object **nil**. The conversion process changes all **NXWriteObjectReference()** calls to **encodeConditionalObject:** messages.

Archiving Objects That You Also Distribute

Use **encodeBycopyObject:** to encode an object for both archiving and distribution. As stated previously, your **encodeWithCoder:** method is used both by the archiving system and the distributed objects system. 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 invokes **encodeObject:** and returns. This way, if **encodeWithCoder:** is passed an NSArchiver as the coder, it archives the object in the usual way. You may find places where you want to use **encodeConditionalObject:** when archiving and use **encodeBycopyObject:** when encoding for distribution. If so, your **encodeWithCoder:** method can check to see if the class of the coder passed to it is an NSArchiver before the method does the encoding.

If you use **encodeBycopyObject:** to archive an object, use **decodeObject** to unarchive it.

Obsolete Archiving Methods

The following table lists obsolete methods you may have implemented to archive or unarchive your objects. The table tells you how to replace these methods if you implemented them.

Obsolete Method TableHeadRule.eps ¬	Possible Replacements
awake TableRule.eps ¬	Append code from awake to initWithCoder:.
finishUnarchiving TableRule.eps ¬	awakeAfterUsingCoder: (in NSObject)
startArchiving: TableRule.eps ¬	classForCoder: (in NSObject) replacementObjectForCoder: (in NSObject) Prepend code from startArchiving: to encodeWithCoder:.

Archiving Mixed Object Graphs

The conversion process changes all of your objects to inherit from NSObject, but your objects might still contain instance variables that inherit from Object. For example, List, HashTable, NXStringTable, and Storage objects may still appear in your code. All of these classes still inherit from Object. This presents a problem for archiving; Objects are archived with **NXWriteObject()** within a **write:** method, but NSObjects are archived with **encodeObject:** within an **encodeWithCoder:** method. Which should you use? The best solution is to perform a deep conversion so that all of the objects in your code inherit from NSObject. If this is not possible, NeXT provides some compatibility methods and functions for you to use when archiving and unarchiving in these situations.

If you have an NSObject subclass with instance variables that inherit from the Object class, use the **encodeNXObject:** and **decodeNXObject** methods to archive and unarchive those objects, as shown in the

following example.

New Code

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [super encodeWithCoder:aCoder];
    [aCoder encodeObject:authorID];
    [aCoder encodeObject:firstName];
    [aCoder encodeObject:lastName];
    [aCoder encodeObject:address];
    [aCoder encodeObject:city];
    [aCoder encodeObject:state];
€€€€[aCoder encodeValuesOfObjCTypes:"i", &contract];
    [aCoder encodeNXObject:titles];
}

- initWithCoder:(NSCoder *)aDecoder
{
    [super initWithCoder:aDecoder];
    authorID €= [[aDecoder decodeObject] retain];
    firstName = [[aDecoder decodeObject] retain];
    lastName €= [[aDecoder decodeObject] retain];
    address €€= [[aDecoder decodeObject] retain];
    city  €€€€=€[[aDecoder decodeObject] retain];
    state  €€€= [[aDecoder decodeObject] retain];
€€€€[aDecoder decodeValuesOfObjCTypes:"i", &contract];
    titles €€€= [[aDecoder decodeNXObject] retain];

    return self;
}
```

Suppose you create a class that inherits from Object and declare some instance variables that are instances of NSObject or one of its subclasses. In this situation, use the **NXWriteNSObject()** and **NXReadNSObject()** functions within the **write:** and **read:** methods, respectively, as shown below.

New Code

```
- write:(NXTypedStream *)stream;
{
    [super write:stream];
    NXWriteNSObject(stream, authorID);
    NXWriteNSObject(stream, firstName);
    NXWriteNSObject(stream, lastName);
    NXWriteNSObject(stream, address);
    NXWriteNSObject(stream, city);
    NXWriteNSObject(stream, state);
€€€€NXWriteTypes(stream, "i", &contract);
    NXWriteObject(stream, titles);
    return self;
}

- read:(NXTypedStream *)stream;
{
    [super read:stream];
    authorID €= [(NSString *)NXReadNSObject(stream) retain];
    firstName = [(NSString *)NXReadNSObject(stream) retain];
    lastName €= [(NSString *)NXReadNSObject(stream) retain];
    address €€= [(NSString *)NXReadNSObject(stream) retain];
    city €€€€€= [(NSString *)NXReadNSObject(stream) retain];
    state €€€€= [(NSString *)NXReadNSObject(stream) retain];
```

```
€€€€NXReadTypes(stream, "i", &contract);
    titles €€€= [NXReadObject(stream) retain];
    return self;
}
```

Restrictions

There are two significant restrictions when archiving objects from both the Object and NSObject world. These are:

SquareBullet.eps ¬There is no sharing of information between the two worlds. Normally, if you archive a complex graph that has cycles where several objects reference a single object, enough information is kept about the objects so that the cycles are detected and objects that are pointed to by many other objects are archived only once. This is still true as long as the graph of objects being archived resides entirely in the Object world or in the NSObject world. In a mixed environment, though, there is no sharing of object information across worlds. Care must be taken not to have cycles in a graph of objects that transcends both worlds.

SquareBullet.eps ¬Container objects (NSArray, NSDictionary, NSValue, etc.) cannot be archived if they contain objects from the other world. Thus, an NSArray cannot be archived if it contains a descendant of Object. Similarly, a List cannot be archived if it contains a descendant of NSObject.

Whenever possible, you should not mix objects from both worlds in your object graphs. Archiving a mixed-world graph of objects will be much slower, take up more space, and be less reliable (due to the lack of object sharing) when compared to archiving a similar graph of objects that all inherit from the same root class.

Summary of Convenience Methods and Functions

The following table summarizes the convenience methods and functions that allow you to archive mixed object graphs.

Function/Method TableHeadRule.eps ¬	Purpose
encodeNXObject: TableRule.eps ¬	To archive an Object appearing in an NSObject.
NXWriteNSObject() TableRule.eps ¬	To archive an NSObject appearing in an Object.
decodeNXObject TableRule.eps ¬	To unarchive an Object appearing in an NSObject.
NXReadNSObject() TableRule.eps ¬	To unarchive an NSObject appearing in an Object.

Archiving Gotchas

After the archiving conversion, watch out for the following:

SquareBullet.eps ¬Always retain all instance variables that are unarchived using **decodeObject**. Never retain any instance variable that is unarchived using the method **decodeValuesOfObjCTypes:"@"**.

SquareBullet.eps ¬Make sure objects that you archive conform to the NSCoding protocol.

SquareBullet.eps ¬NXColor structures are replaced by NSColor objects in OpenStep. (If you're converting in stages, this won't happen until stage 3.) Use **decodeNXColor** to unarchive NXColor structures and **decodeObject** to unarchive NSColor objects. For more information, see the section ªColor Conversionº in the chapter ªConverting Application Kit Classes.º

Defaults Conversion

Stage 5

The Foundation Framework provides a new user defaults system managed by an object of class NSUserDefaults. This section provides a brief overview of the new system. You can learn more by reading the NSUserDefaults class specification in the *Foundation Framework Reference*.

In this new system, defaults are stored in separate domains. Within each domain, the defaults are stored in NSDictionary objects. Basically, you perform two functions with NSUserDefaults:

SquareBullet.eps ¬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.

SquareBullet.eps ¬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

In most cases, the conversion process changes all accesses to the defaults database automatically. If you use any of the functions shown in the following table, you may have to perform some conversion yourself. For these functions, the conversion process displays warning messages that will help you decide what to do. The table gives you a basic idea of what to replace your function call with.

Obsolete Function TableHeadRule.eps ¬	**Possible Replacement**
NXWriteDefaults() TableRule.eps ¬	setObject:forKey:
NXRegisterDefaults() TableRule.eps ¬	registerDefaults:
NXSetDefaultsUser() TableRule.eps ¬	initWithUser:

Defaults Gotchas

After the Defaults Conversion, watch out for the following:

NXSetDefaultsUser()

The conversion process changes your call to **NXSetDefaultsUser()** to the **initWithUser:** method. When you call **NXSetDefaultsUser()**, it changes the defaults database that is affected when you call any other defaults functions. However, **initWithUser:** does not. **initWithUser:** is sent only to your own instance of NSUserDefaults; if you then invoke **standardUserDefaults**, it returns the globally shared instance of NSUserDefaults. The shared instance does not reflect the changes you made to the instance initialized with **initWithUser:**.

Foundation Conversion

Stage 1

The Foundation conversion introduces the basic Foundation classes into your code: notably NSArrays and NSExceptions.

NSArray Replaces List

Wherever you used to see a List object in the Application Kit, you now see either an NSArray or an NSMutableArray object. For example, the View class's subviews, Matrix's **cellList**, and Application's **windowList** are now stored as NSArrays. (Since they are no longer Lists, the names **cellList** and **windowList** have changed to **cells** and **windows**, respectively.)

As described previously in the section ªClass Clusters,º NSArray is a class cluster defined in the Foundation Framework. NSArray objects can't be modified, but NSMutableArray objects can.

The Foundation conversion converts only Lists returned by the Application Kit or passed to the Application Kit. If your application has a List that does not interact with an Application Kit object, it will remain a List. After all six required conversions are complete, you may choose to run the **ListToMutableArray.tops** conversion script to eliminate all List objects from your code. For more information, see the section ªList To MutableArray Conversion.º

NSArrays are essentially the same as Lists. There are some important differences, though. To learn about these differences, see the section ªList To MutableArray Conversion.º

Changes to Exception Handling

The Foundation Framework has a new class, NSException, which you use during exception handling. The NSException object describes the exceptional condition. Exception handlers are similar to the old exception handlers, but you raise an exception differently. (The names of macros and functions have changed slightly,

but those are taken care of automatically.) The **NX_RAISE()** macro is now obsolete. When you compile your code after this conversion, each occurrence of **NX_RAISE()** will be flagged with a message telling you to use an NSException object instead.

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

SquareBullet.eps ¬NSException uses names rather than numbers to identify exceptions.
SquareBullet.eps ¬You always associate an error message with an NSException.
SquareBullet.eps ¬You provide any necessary application-specific data through an NSDictionary.
SquareBullet.eps ¬You use the variable **localException** where you used to use **NXLocalHandler**.
SquareBullet.eps ¬You pass a single argument, the NSException object, to functions such as **NSSetUncaughtExceptionHandler()**.

NSDictionary is a new class in Foundation that stores key-value pairs. Like NSArray, NSDictionary is a class cluster with two public superclasses: NSDictionary (unmodifiable) and NSMutableDictionary (modifiable).

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.

You should convert all calls to **NX_RAISE()** to code that creates an NSException object and raises the exception it represents as shown in the following example.

Old Code

```
int returnValue;
float variable;
...
returnValue = aFunction(variable);
if (returnValue)
    NX_RAISE(AFUNCTON_ERROR, returnValue, variable);
```

New Code

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

This message supplies the same information to the exception as the **NX_RAISE()** macro did. To be able to store the values of **returnValue** and **variable** in an NSDictionary, you need to convert them to objects. NSNumber, described previously, was created for just this purpose. Within the error handler itself, you use the **localException** variable to refer to the exception. Use the message **[[localException userInfo] objectForKey:@"Return Value"]** to retrieve the return value from the dictionary.

The macro that performs assertions (**NX_ASSERT()**, now **NSAssert1()**) has been modified as well. Previously, **NX_ASSERT()** just printed a message if the assertion failed. Now it raises the NSInternalInconsistencyException. It raises this exception by first sending a message to an object of type NSAssertionHandler, which is a new class in Foundation Framework. You can use NSAssertionHandler directly if you want to provide more detailed control over failed assertions, specifically for multithreaded applications. For more information, see the NSAssertionHandler class specification in the *Foundation Framework Reference*.

The following table summarizes the changes to exception handling.

Obsolete Macro or Variable TableHeadRule.eps ¬	Replacement
NX_RAISE() macro TableRule.eps ¬	exceptionWithName:reason:userInfo: (or any other NSException +*classname* method)
NXLocalHandler constant TableRule.eps ¬	localException constant

Foundation Conversion Gotchas

After the Foundation conversion has been run on your code, look out for the following things:

Link Enumerators in DataLinkManager

Foundation's NSEnumerator object replaces DataLinkManager's link enumerator type. The NSEnumerator object numerates collections of objects. It provides an easy way to step through each item in an unordered collection, such as an NSDictionary. You no longer have to prepare the link enumerator's state, so the methods **prepareEnumerationState:forLinksOfType:** and **nextLinkUsing:** are obsolete. To retrieve an NSEnumerator from the NSDataLinkManager, use either **sourceLinkEnumerator** or **destinationLinkEnumerator**. You can walk through the list of links using NSEnumerator's **nextObject** method.

Another change is NXDataLink's **lastUpdateTime** method. This method now returns an NSDate object instead of **time_t**. NSDate is a new class that represents specific point in time, both a date and a time. Like NSString, NSDate was created to be operating-system-independent and to make internationalization easier. For more information on NSDate, see the *Foundation Framework Reference*.

Finding the Path for Resources in NSBundle

A bundle's resources are now one level deeper inside the bundle than they previously were. All bundles now have a **Resources** directory that contains all resources, including localized resources. If you used the **bundlePath** method to return the location of a resource, your code is now incorrect. Replace the **bundlePath** message with **resourcePath**.

Hash And String Table Conversion

Optional

Run the **HashAndStringTableConversion.tops** script to convert your HashTable and NXStringTable objects to NSDictionary objects. This script is optional; if you don't use this conversion, your program will still compile and run. If you're performing a shallow conversion and you want to keep your HashTable objects, be sure to explicitly import **<objc/HashTable.h>**. **AppKit.h** no longer imports **HashTable.h**.

Note: To learn how to run a single **tops** script, see the on-line release note **/NextLibrary/Documentation/NextDev/ReleaseNotes/ConvertingYourCode.rtf**. Don't run the **HashAndStringTableConversion.tops** script until all six conversion stages are complete.

An NSDictionary is an object that stores key-value pairs. An NSDictionary differs from a HashTable in the following ways:

SquareBullet.eps ¬ The key and the value must be objects.
SquareBullet.eps ¬ You can store values of different classes in an NSDictionary object.
SquareBullet.eps ¬ You cannot add **nil** keys or **nil** values to NSDictionary objects.

All of these differences have consequences when converting. Especially for HashTable objects, you may need to perform some conversion yourself after you run the script.

There are actually two constructs in OpenStep that you may use in place of a HashTable object. One is an NSDictionary object, and the other is an NSMapTable structure. If your HashTable uses C strings, NXAtoms,

or objects as keys, you can easily convert it to an NSDictionary, and the conversion script helps you do this. If you use another type of key, convert the HashTable to an NSMapTable structure as described in the section ªConverting HashTables to NSMapTables.º The conversion script does not help you convert HashTables to NSMapTables. Like all objects, NSDictionary objects can be archived and distributed, and they use the OpenStep memory allocation and deallocation scheme. For these reasons, you will probably want to use NSDictionary objects instead of NSMapTable structures wherever possible.

Creating NSDictionary Objects

The conversion script will change the initialization method you use to a corresponding initialization method for NSDictionary. It is recommended that you change this to a ***+classname*** method so that the NSDictionary will be autoreleased as well as allocated and initialized. The following table lists and describes the ***+classname*** methods that create NSDictionary objects.

NSDictionary Method **TableHeadRule.eps ¬**	**Creates:**
dictionary TableRule.eps ¬	An empty NSDictionary.
dictionaryWithContentsOfFile: TableRule.eps ¬	An NSDictionary with the contents of the specified ASCII property-list file.
dictionaryWithObjectsAndKeys: TableRule.eps ¬	An NSDictionary with keys and values listed in a comma-separated, nil-terminated list of key-value pairs.
dictionaryWithObjects:forKeys: TableRule.eps ¬	An NSDictionary with values listed in the first argument corresponding to the keys listed in the second argument.
dictionaryWithObjects:forKeys:count: TableRule.eps ¬	Like dictionaryWithObjects:forKeys:, but you specify a count for efficiency.
dictionaryWithCapacity:	An empty NSMutableDictionary with enough space to store the specified number of key-value pairs. (NSMutableDictionary only).

TableRule.eps ¬

Reading NXStringTables from a .strings File

Previously, you stored the entries for the NXStringTable in a **.strings** file on disk and read that into the NXStringTable object with the method **readFromFile:** or **newFromFile:**. In OpenStep, **.strings** files are read into NSString objects and then converted to NSDictionary objects with the **propertyListFromStringsFileFormat** method. The NSDictionary is converted back to an NSString before it is written out to the **.strings** file. The following example illustrates this conversion.

Old Code

```
myStringTable = [NXStringTable newFromFile:"myFile.strings"];
...
[myStringTable writeToFile:"myFile.strings"];
```

New Code

```
myStringTable = [[NSString
     stringWithContentsOfFile:@"myFile.strings"]
     propertyListFromStringsFileFormat];
...
[[myStringTable descriptionInStringsFileFormat]
     writeToFile:@"myFile.strings" atomically:YES];
```

Converting HashTables to NSDictionary

If your HashTable's keys are C strings, NXAtoms, or objects, convert the HashTable to an NSDictionary. You need to convert the values to objects manually if you were not already using objects.

If the values were C strings or NXAtoms, convert them to NSStrings. If you used another type of value, convert them to NSNumber or NSValue objects. NSNumber and NSValue are Foundation classes that allow you to store Objective-C types in objects. Use NSNumber for simple number types (such as **int**, **float**, or **BOOL**). Use NSValue for complex types such as structures. To learn more about these classes, see the *Foundation Framework Reference*.

Old Code

```
int intValue;
void *ptrValue;
...
intHash = [[HashTable alloc] initKeyDesc:"*"
        valueDesc:"i"];
ptrHash = [[HashTable alloc] initKeyDesc:"*"
        valueDesc:"!"];
...
[intHash insertKey:"int1"
     value:&intValue];
[ptrHash insertKey:"ptr1"
     value:&ptrValue];
```

New Code

```
int intValue;
void *ptrValue;
...
intHash = [NSMutableDictionary dictionary];
ptrHash = [NSMutableDictionary dictionary];
...
[intHash setObject:[NSNumber numberWithInt:intValue]
     forKey:[NSString stringWithCString:"int1"]];
```

```
[ptrHash setObject:[NSValue value:ptrValue
     withObjCType:@encode(void *)]
     forKey:[NSString stringWithCString:"ptr1"]];
```

Converting HashTables to NSMapTables

The **HashAndStringTableConversion.tops** works well only for objects that use strings or objects for keys. If you use something else for keys, convert the HashTable manually to an NSMapTable structure. Like a HashTable, the NSMapTable structure stores key-value pairs and it lets you specify the types of keys and values you are going to store. You can choose among integers, pointers, and objects. There are functions that perform essentially the same operations on the NSMapTable as the HashTable methods perform. The following example shows how a HashTable using integer keys would look before and after conversion.

Old Code

```
int intValue;
id anObject;
...
intKeyHash = [[HashTable alloc] initKeyDesc:"i"];
...
[intHash insertKey:(void *)intValue value:anObject];
...
[intKeyHash free];
```

New Code

```
int intValue;
id anObject;
...
intKeyHash = NSCreateMapTable(NSIntMapKeyCallBacks,
```

```
     NSObjectMapValueCallBacks, NULL);
...
NSMapInsert(intKeyHash, (void *)intValue, anObject);
...
NSFreeMapTable(intKeyHash);
```

Removing and Freeing Objects in an NSMutableDictionary

A HashTable lets you free the objects it was storing using **freeObjects** or **freeKeys:values:**. These methods remove the specified objects from the HashTable and then free them, leaving the HashTable still allocated.

To perform the analogous function with an NSMutableDictionary, simply remove the object from the NSMutableDictionary. When you add an object to the NSMutableDictionary, it makes a copy of the key and retains the object used as the value. When you remove the key-value pair from the NSMutableDictionary, it releases both the key and the value. When an object is released, its reference count is decremented. The object is deallocated when its reference count is 0.

The following table shows the methods that remove and free objects from a HashTable, shows the analogous methods for NSMutableDictionary, and explains what's different about the NSMutableDictionary method.

HashTable Method TableHeadRule.eps ¬	**NSMutableDictionary Method**	**Differences**
empty	removeAllObjects	Keys and values are released and therefore may be deallocated as a side effect of this method.
TableRule.eps ¬		
removeKey:	removeObjectForKey:	Both key and value are released and therefore may be deallocated as a side effect of this method.

TableRule.eps ¬

freeObject:	removeObjectForKey:	Both key and value are released. Releasing does not necessarily deallocate the object.

TableRule.eps ¬

freeKeys:values:	removeAllObjects	Both the keys and their values are released. Releasing does not necessarily deallocate the objects.

TableRule.eps ¬

Stepping Through an NSDictionary

If you want to access each entry in a HashTable one by one, you create an NXHashState variable and use that variable to retrieve the next key-value pair in the HashTable.

To access each entry in a NSDictionary, you use an NSEnumerator object. The NSEnumerator can access each key in the NSDictionary. To step through each entry in the dictionary, you send the NSEnumerator the message **nextObject** and use **objectForKey:** to retrieve the value.

If you have a function or method that merges two HashTables, you could convert it to a function that merges two NSMutableDictionary objects as shown in the example below. (The conversion script makes some of these changes for you.)

Old Code

```
void mergeHashs(HashTable *hash1, HashTable *hash2)
{
     NXHashState state = [hash1 initState];
     void *key, *value;
     while([hash1 nextState:&state key:&key value:&value])
```

```
        [hash2 insertKey: key value: value];
}
```

New Code

```
void mergeHashs(NSDictionary *hash1,
     NSMutableDictionary *hash2)
{
     NSEnumerator *state = [hash1 keyEnumerator];
     NSString *key;
     key = [state nextObject];
     while(key) {
         [hash2 setObject:[hash1 objectForKey:key] forKey:key];
         key = [state nextObject];
     }
}
```

readFromFile: Conversion

The **HashAndStringTableConversion.tops** script makes the following conversion for the NXStringTable method **readFromFile:**.

Old Code

```
[anyVar readFromFile:aFile];
```

New Code

```
[anyVar addEntriesFromDictionary:[[NSString
     stringWithContentsOfFile:[NSString stringWithCString:aFile]]
```

```
        propertyListFromStringsFileFormat]];
```

The script does not first check to make sure **anyVar** is an NXStringTable object. If you defined **readFromFile:** in a custom class, you will need to look for occurrences of this message and change it back.

List To MutableArray Conversion

Optional

Run the conversion script **ListToMutableArray.tops** to change all occurrences of Lists in your code to NSMutableArrays. As explained previously, the Foundation conversion changes only Lists that interact with Application Kit objects. Run this script to convert all remaining List objects. This script is optional; if you don't use it, your code will still compile and run. If you're performing a shallow conversion and you want to keep your List objects, be sure to explicitly import **<objc/List.h>**. **AppKit.h** does not import this header anymore.

Note: To learn how to run a single **tops** script, see the on-line release note **/NextLibrary/Documentation/NextDev/ReleaseNotes/ConvertingYourCode.rtf**. Don't run this script until all six conversion stages are complete.

The main differences between List and NSArray objects are:

SquareBullet.eps ¬Adding **nil** to an NSArray raises an exception.
SquareBullet.eps ¬Indexing an NSArray out of bounds raises an exception.

These and other differences are explained in this section.

Invalid Indexes

When NSArray receives an invalid index, it raises an exception rather than returning **nil**. Therefore, you

should never test that the return value of a method such as **objectAtIndex:** is **nil**. For example, you might have a loop that accesses each object in a List, as shown below. After this code is converted, it raises an exception where you want it to exit the loop. To access each object in an NSArray one by one, you should instead use the for loop shown below.

Old Code

```
int i;
List *myList = [myForm cellList];
id anObject;
...
i = 0;
while (anObject = [myList objectAt:i]) {
     /* do something here */
     i++;
}
```

New Code

```
int i, cellCount;
NSArray *myList = [myForm cells];
id anObject;
...
for (i = 0, cellCount = [myList count]; i < cellCount; i++) {
     anObject = [myList objectAtIndex:i];
     /* do something here */
}
```

Deallocating NSArrays

Previously, when you freed a List, the objects contained in the List were not freed. When you release an NSArray (or any other Foundation collection object), the objects contained in the array are released as well as the NSArray itself. If you have code that accesses a list element after the List is freed, it will produce a run-time error after the conversion. You should change it as shown below.

Old Code

```
anObject = [myList objectAt:0];
[myList free];
if ([anObject intValue] == 0)
     ...
```

Bad New Code

```
anObject = [myList objectAtIndex:0];
[myList release]; /* myList is now an NSArray */
if ([anObject intValue] == 0)
     /* Runtime error! anObject was released */
     ...
```

Good New Code

```
anObject = [[myList objectAtIndex:0] retain];
[myList release];
if ([anObject intValue] == 0)
     ...
```

Creating NSArrays

When you create a List, you use either the **new** method or **alloc** and **init**. As explained previously, the

recommended way to create an OpenStep object is to use the ***+classname*** method. ***+classname*** methods allocate, initialize, and autorelease the object. NSArray and NSMutableArray offer the methods listed in the following table to allocate and initialize autoreleased instances. Change your **alloc** and **init** or **new** method to one of these.

NSArray Method	**Creates:**
TableHeadRule.eps ¬	
array	An empty NSArray.
TableRule.eps ¬	
arrayWithObject:	An NSArray containing one object.
TableRule.eps ¬	
arrayWithObjects:	An NSArray containing the specified list of objects. The list must be terminated with **nil**.
TableRule.eps ¬	
arrayWithContentsOfFile:	An NSArray containing the objects in the specific file.
TableRule.eps ¬	
arrayWithCapacity:	An empty NSMutableArray with enough space for the specified number of objects. (NSMutableArray only.)
TableRule.eps ¬	

Setting the Capacity

The List class provides methods that allow you to set a List's capacity. NSMutableArray does not provide such a method. However, with the **arrayWithCapacity:** method, you can specify an initial size when you create the NSMutableArray. You can use this method to allocate all or most of the memory at one time, which is more efficient than allocating memory each time an object is added.

List Method	**Possible Replacement in NSMutableArray**

TableHeadRule.eps ¬

setAvailableCapacity: TableRule.eps ¬	arrayWithCapacity: (to create the NSMutableArray)
capacity TableRule.eps ¬	count

Void Methods

Like all other OpenStep objects, NSArrays return void when they have no meaningful value to return. The conversion script does not catch all of the places where a List method returned a value but an NSArray returns **void**. If you use the value returned by the List method, you'll receive a compiler error. You should split the NSArray method in two as shown below.

Old Code

```
anObject = [myList removeLastObject];
```

Bad New Code

```
anObject = [[myList lastObject] retain];
[myList removeLastObject];
```

NSArray or NSMutableArray

The script **ListToMutableArray.tops** changes all List objects in your application to NSMutableArray objects. Because NSArrays are more memory efficient than NSMutableArrays, you should use NSArrays wherever possible. You should examine each NSMutableArray in your application. If you never send it **addObject:** or **removeObject:** (or you can consolidate all **addObject:** messages into an **arrayWithObjects:** message), you should

change the NSMutableArray into an NSArray.

Archiving NSArrays

NSArray and other container objects cannot be archived if they contain objects that inherit from Object rather than NSObject. Similarly, a List cannot be archived if it contains a descendant of NSObject.

Notification Conversion

Stage 5

The Foundation Framework introduces a notification system, which is a way for objects that don't know about each other to communicate. Every application now 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 advantages:

SquareBullet.eps ¬Any number of objects may receive the notification, not just the delegate object.
SquareBullet.eps ¬An object may receive any message you like from the notification center, not just the predefined delegate methods.
SquareBullet.eps ¬The object posting the notification does not even have to know that the other object exists.

Some Application Kit objects that use delegates now use the notification system to inform delegates that an event occurred. 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. 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 use the notification system in your application and how the use of the

notification system affects 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.

New Code

```
[[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), an associated object, and sometimes extra information stored in a userInfo dictionary. Your method can use the **[notification object]** message to retrieve the

associated object, which is typically the object that posted the notification, and can retrieve other necessary information from the userInfo dictionary using **[notification userInfo]**.

For more information, see the NSNotificationCenter and NSNotification class specifications in the *Foundation Framework Reference*.

Changes to Delegates

Although Application Kit objects that send messages to delegates now 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.

Delegate Methods Implementations

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. The conversion process makes this change for you.

Old Code

```
- windowDidUpdate:sender
{
    if ([sender isMainWindow])
        [view updateLinksPanel];
    return self;
}
```

New Code

```
- (void)windowDidUpdate:(NSNotification *)notification
```

```
{
     NSWindow *theWindow = [notification object];
            /* the "sender" */

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

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 TableHeadRule.eps ¬	**Replacement**
textDidChange: TableRule.eps ¬	controlTextDidBeginEditing:
textDidEnd:endChar: TableRule.eps ¬	controlTextDidEndEditing:
textDidGetKeys:isEmpty: TableRule.eps ¬	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

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

Subclasses of Classes with Delegates

Application Kit objects with delegates often have their own methods that correspond to and invoke the delegate methods. For example, the **windowMoved:** method in Window was invoked when the user moved a window, and it invoked the delegate method **windowDidMove:**. When you subclass an Application Kit object with a delegate, you override such methods if you want your subclass to respond differently than the default case.

Many such methods in the classes NSApplication, NSWindow, NSSplitView, and NSText are obsolete. The

conversion process marks these methods with a warning message that tells you what notification your subclass should register to receive. The section ªUsing the Notification Systemº earlier in this chapter shows you how to register to receive a notification.

Obsolete Delegate Methods

The disk mounting and power-off functions that used to be part of the Application class are now part of the NSWorkspace class. (The NSWorkspace class is new in OpenStep.) NSWorkspace does not have a delegate, but it does post notifications about these events.

If your application delegate responded to any of the methods listed in the following table, it should now register to receive the corresponding notification. The section ªUsing the Notification Systemº earlier in this chapter shows you how to register to receive a notification.

Obsolete Delegate Method TableHeadRule.eps ¬	**Possible Replacement**
app:applicationWillLaunch: TableRule.eps ¬	Register for NSWorkspaceWillLaunchApplicationNotification.
app:applicationDidTerminate: TableRule.eps ¬	Register for NSWorkspaceDidTerminateApplicationNotification.
app:fileOperationCompleted: TableRule.eps ¬	Register for NSWorkspaceDidPerformFileOperationNotification.
app:mounted: TableRule.eps ¬	Register for NSWorkspaceDidMountNotification.
app:powerOffIn:andSave: TableRule.eps ¬	Register for NSWorkspaceWillPowerOffNotification.
app:unmounting: TableRule.eps ¬	Register for NSWorkspaceWillUnmountNotification.
app:unmounted: TableRule.eps ¬	Register for NSWorkspaceDidUnmountNotification.

appAcceptsAnotherFile: TableRule.eps ¬	None needed.

Other Notifications Posted by Application Kit Objects

In addition to delegate methods, there are other places where Application Kit objects use notifications to notify objects of events. In a few cases, the posting of a notification makes an Application Kit method obsolete. The following table lists methods that are obsolete because the object now posts a notification and what you should use as a replacement.

Obsolete Method **TableHeadRule.eps ¬**	**Possible Replacement**
colorListDidChange: (in NXColorList) TableRule.eps ¬	Register for NSColorListDidChangeNotification.
updateCustomColorList (in NXColorPanel) TableRule.eps ¬	Register for NSColorListDidChangeNotification.
updateColorList (in NXColorPicker) TableRule.eps ¬	Register for NSColorListDidChangeNotification.
descendantFrameChanged: (in NSView) TableRule.eps ¬	Register for NSViewFrameDidChangeNotification.
notifyAncestorWhenFrameChanged: (in NSView) TableRule.eps ¬	Register for NSViewFrameDidChangeNotification.
suspendNotifyAncestorWhenFrameChanged: (in NSView) TableRule.eps ¬	setPostsFrameChangedNotifications:NO
screenChanged: (in NSWindow) TableRule.eps ¬	Register for NSWindowDidChangeScreenNotification.
windowExposed: (in NSWindow) TableRule.eps ¬	Register for NSWindowDidExposeNotification.
windowMoved: (in NSWindow)	Register for NSWindowDidMoveNotification.

TableRule.eps ¬

Notification Gotchas

After the notification conversion, look out for the following:

Pseudo-Delegates

The notification conversion changes any invocation of a delegate method to a message that posts the notification that invokes the delegate method. If you implemented a ªpseudo-delegate,º this causes delayed recursion, which you won't be able to detect until run time.

For example, suppose you have two objects that need to act as a Window's delegate. You assign one of the objects to be the actual delegate, and inside its delegate methods, you invoke the second object's (the ªpseudo-delegate'sº) methods. The code samples below show the code for the true delegate's **windowDidBecomeKey:** method before and after conversion. Because this method is invoked every time NSWindowDidBecomeKeyNotification is posted, it is now in an infinite loop.

Old Code

```
- windowDidBecomeKey: sender
{
      ...
      [wannaBeDelegate windowDidBecomeKey: sender];
      return self;
}
```

Bad New Code

```
- (void)windowDidBecomeKey:(NSNotification *)notification
{
```

```
    NSWindow *theWindow = [notification object];
    ...
    [[NSNotificationCenter defaultCenter] postNotificationName:
        NSWindowDidBecomeKeyNotification object: theWindow];
    /* BAD CODE! This message causes delayed recursion. */
}
```

To correct this problem, remove the **postNotificationName:object:** message from the **windowDidBecomeKey:** method. Register **wannaBeDelegate** to receive the NSWindowDidBecomeKeyNotification in another place. These fixes are shown below.

Good New Code

```
@implementation WannaBeDelegate
- init
{
    ...
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(windowDidBecomeKey:)
        name:NSWindowDidBecomeKeyNotification
        object:nil];
    ...
}
...
@end

@implementation OfficialDelegate
...
- windowDidBecomeKey:(NSNotification *)notification
{
    NSWindow *theWindow = [notification object];
```

```
        ...
}
...
@end
```

Rect Conversion

Stage 1

The NXRect and NXSize structures, now NSRect and NSSize, have moved to the Foundation Framework. They are now defined in the header file **NSGeometry.h**. Other than this, the structures themselves remain the same.

You will see two important differences in using these structures: All Application Kit functions and methods that used to take the address of an NXRect or an NXSize now take the value of the structure. Similarly, all Application Kit functions and methods that used to return a pointer to an NXRect or NXSize now return the structure itself. This change was made to eliminate the aliasing problems that can occur when you pass the address of a structure.

The conversion process makes these changes for you in most cases. (In places where you passed NULL for one of these structures, the conversion process changes NULL to **NSZeroRect** or **NSZeroSize**.) The rest of this section describes areas of the Application Kit where these changes require you to perform some of the conversion yourself.

Functions That Draw Borders

The functions that draw borders, **NXDrawButton()**, **NXDrawGrayBezel()**, **NXDrawGroove()**, **NXDrawTiledRects()**, and **NXDrawWhiteBezel()**, have changed slightly. You pass one of these functions two rectangles, the bounds

rectangle and the clipping rectangle. The function draws a border around the part of the bounds rectangle that intersects with the clipping rectangle. Previously, to draw the border around the entire bounds rectangle, you would pass NULL as the second argument. Now you can't do this, so you have to make sure the intersection of the two rectangles is the bounds rectangle. A rectangle intersected with itself is that rectangle. Thus, to have one of these functions draw a border around the entire bounds rectangle, you make the change shown below. The conversion process makes this change for you, but it flags each case so that you can verify that the invocations still make sense.

Old Code

```
NXDrawButton(&boundsRect, NULL);
```

New Code

```
NSDrawButton(boundsRect , boundsRect);
```

constrainFrameRect:toScreen: in Window

The Window class's **constrainFrameRect:toScreen:** method used to return a **BOOL** value that indicated whether the first argument, the frame rectangle, was modified. It now returns the frame rectangle so that it conforms with other methods that perform similar functions. You will need to change the variable that captured this method's return value, and you will probably need to add code that checks whether this rectangle was modified, as shown in the following example.

Old Code

```
NXRect *myFrameRect;
NXScreen *myScreen;
BOOL frameChanged;
...
```

```
frameChanged = [myWindow constrainFrameRect:myFrameRect
     toScreen:myScreen];
if (frameChanged) ...
```

New Code

```
NSRect myFrameRect;
NSScreen *myScreen;
NSRect newFrameRect;
...
newFrameRect = [myWindow constrainFrameRect:myFrameRect
     toScreen:myScreen];
if (!NSEqualRects(newFrameRect, myFrameRect)) ...
```

Boundary Rectangles in Cell

The Cell methods **getIconRect:**, **getTitleRect:**, **getDrawRect:**, and **calcCellSize:inRect:** have been changed to **drawingRectForBounds:**, **titleRectForBounds:**, **imageRectForBounds:**, and **cellSizeForBounds:** in NSCell to be more clear. Each of these new methods returns an NSRect, leaving the NSRect passed in as an argument unchanged. The conversion process changes these messages as shown in the following example.

Old Code

```
[aCell getTitleRect:&titleRect];
```

New Code

```
titleRect = [aCell titleRectForBounds:titleRect];
```

Also in NSCell, you can no longer pass NULL as the NSRect for the method **trackMouse:inRect:ofView:**. The

conversion process flags places where you have done this, and you must change NULL to the cell's actual boundary yourself.

Boundary Rectangles in View

Previously, if you wanted to redraw only a portion of a View, you passed an array of three NXRects to one of the View's display methods. The first NXRect indicated the enclosing rectangle of the second and third NXRects, and the method drew in the union of the last two NXRects. You used **calcUpdateRects:::** to calculate the area to be redrawn. However, the usual use of the display methods was to redraw a single rectangle.

To simplify the API, the display methods now take a single NSRect that indicates the area to be redrawn. The different display methods have been consolidated into three: **display**, **displayRect:**, and **drawRect:** (which replaces **drawSelf::**). The **calcUpdateRects:::** method is obsolete because it is no longer necessary. If you have subclassed View and overridden any of these methods, you must rewrite your code. The following table summarizes the changes made to the View display methods.

View Method	**Replacement**
TableHeadRule.eps ¬	
calcUpdateRects:::	None necessary
TableRule.eps ¬	
display	display
TableRule.eps ¬	
display::	displayRect:
TableRule.eps ¬	
display:::	displayRect:
TableRule.eps ¬	
displayFromOpaqueAncestor:::	displayRect:
TableRule.eps ¬	
drawSelf::	drawRect:
TableRule.eps ¬	

Storage Conversion

None of the scripts converts Storage objects. If you use Storage objects in your application, you may choose either not to convert them or to do the following:

1. Convert the Storage elements into objects. You can do this either by defining a new class or by using NSValue objects.
2. Convert the Storage object itself into an NSArray object.

The following example shows how you might convert a Storage object into an NSMutableArray.

Old Code

```
Graph graph;
...
graphs = [Storage newCount:0
     elementSize:sizeof(Graph)
     description:@encode(Graph)];
...
[graphs addElement:&graph];
```

New Code

```
Graph graph;
...
graphs = [NSMutableArray array];
...
```

```
[graphs addObject:[NSValue value:&graph
     withObjCType:@encode(Graph)]];
```

Stream Conversion

Stage 2

Application Kit objects and functions now use OpenStep objects where they used to use streams or **void *** pointers. The Stream conversion replaces all Application Kit uses of streams or **void *** pointers with the objects listed in the table below. You may have to complete the conversion by changing some of your variables from typed streams to the appropriate object. You also may have to rewrite code that manipulates data previously stored in a stream to use the methods provided by the object.

Streams for: **TableHeadRule.eps ¬**	**Are replaced by:**
Archiving/Typed streams TableRule.eps ¬	NSArchiver, NSUnarchiver
ASCII text TableRule.eps ¬	NSString
NSCell TableRule.eps ¬	NSString
NSGetWindowServerMemory TableRule.eps ¬	NSString
NSPasteboard filenames TableRule.eps ¬	NSArray of NSStrings
NSPasteboard other data TableRule.eps ¬	NSData

NSSelection TableRule.eps ¬	NSData
NSText TableRule.eps ¬	NSString
PostScript or EPS code TableRule.eps ¬	NSData
RTF and RTFD text TableRule.eps ¬	NSData
Word tables TableRule.eps ¬	NSData

This script converts only streams returned by the Application Kit or passed to the Application Kit. If your application has a stream that does not interact with an Application Kit object, it will not be changed. After all six required conversions are complete, you may choose to run the **StreamToMutableData.tops** or **StreamToString.tops** conversion script to eliminate all streams from your code. For more information, see the section ªOptional Stream Conversions.º

Optional Stream Conversions

Optional

To remove the use of streams from your code, you have two choices of scripts to run. If your streams contain textual information, run **StreamToString.tops**, which converts your streams to NSString objects. If your streams are not textual, run **StreamToMutableData.tops**, which converts streams to NSMutableData objects.

You may have to complete the conversion by changing some of your variables from streams to the appropriate object. You also may have to rewrite code that manipulates data previously stored in a stream to use the methods provided by the object.

Note: To learn how to run a single **tops** script, see the on-line release note **/NextLibrary/Documentation/NextDev/ReleaseNotes/ConvertingYourCode.rtf**. Don't run these scripts until all six conversion stages are complete.

String Conversion

Stage 1

In the Application Kit, functions and methods that previously took C string arguments now take NSString objects. The NSString class comes from the Foundation Framework. Like NSArray, NSString is a class cluster with two public superclasses: NSString (not modifiable) and NSMutableString (modifiable).

The Application Kit uses NSStrings because they make internationalization easier and make your code more portable. NSStrings store character strings in a representation-independent format, which means you don't have to know about your string's encoding when you are programming. In addition, NSStrings are allocated and deallocated the same way that other OpenStep objects are, which makes memory allocation for strings much less error prone.

Although the actual representation of character strings stored in NSStrings is independent of any particular implementation, in general you can think of the contents of NSString objects as being, canonically, Unicode characters (defined by the **unichar** data type). Methods that use the terms ªcharacter,º ªrange,º and ªlength,º refer to strings of **unichar**s and ranges and lengths of **unichar** stringsÐthis is important, because conversion between **unichar**s and other character encodings is not necessarily one to one. For instance, a NEXTSTEP encoded string of a given length might contain fewer or more characters when encoded as **unichar**s. Another important point is that **unichar**s don't necessarily correspond one to one with what is normally thought of as ªlettersº in a string; if you need to go through a string in terms of ªletters,º use the NSString method **rangeOfComposedCharacterSequenceAtIndex:**.

Any function that you are able to perform on a C string you can now perform with an NSString method. An

associated class, NSScanner, lets you extract numbers and strings from an NSString, just as the **sscanf()** function lets you extract values from a C string. In addition, NSString has many convenient methods that make it easy to work with UNIX pathnames.

The NSString class cluster description in the *Foundation Framework Reference* contains a complete description of how to use NSStrings. The rest of this section describes the differences you'll see regarding strings and NSStrings once the string conversion is complete.

Automatic NSString Conversion

The following table shows examples of the basic changes that the required String conversion makes to your strings. The conversion process preserves your C string variables using the NSString methods **stringWithCString:** and **cString** so you don't have to convert the variables to NSStrings. The **stringWithCString:** method creates an NSString object with the same value as the C string variable supplied to it. Conversely, the **cString** method creates a C string from an NSString object.

Old Code	**Converted Code**
TableHeadRule.eps ¬	
[NXApp loadNibSection:nibString...]	[NSApp loadNibSection: €€€€[NSString stringWithCString:nibString]...]
TableRule.eps ¬	
[NXApp loadNibSection:"My.nib"...]	[NSApp loadNibSection:@"My.nib"] ...];
TableRule.eps ¬	
titleString = [window title];	titleString = [[window title] cString];
TableRule.eps ¬	

After you have completed all six required conversion stages, you may choose to run the script **StringConversion2.tops**. This script eliminates some of the excess **stringWithCString:** and **cString** messages by

converting the C string in question to an NSString. For more information, see the section ªOptional String Conversion.º

The **cString** method converts the NSString to a C string using the system's default encoding. If it can't create a C string in the default encoding, it raises an exception. You should look at each occurrence of **cString** in your code after the conversion to see if this is a potential problem. If it is, you have a choice of how to fix it:

SquareBullet.eps ¬ You can convert the **char *** variable to an NSString, thus eliminating the need for the **cString** method. NeXT provides some extensions to the OpenStep specification to allow you to keep some information as an NSString. For example, if you are performing file manipulations, you can use the NSFileManager object, which is a NeXT extension, so that you can store filenames in NSString objects instead of passing C string arguments to UNIX system calls. NSFileManager is discussed more in the section ªConverting for Complete Localizability.º

SquareBullet.eps ¬ You can use the **lossyCString** method in place of **cString**. When **lossyCString** encounters a character that can't be represented, it converts it to a character that can be represented. For example, it might convert an em dash (Ð) to a dash (-) or a bullet (·) to an asterisk (*). Use **lossyCString** when you need to convert to a C string and you don't need the conversion to be absolutely correct.

String Variables

The conversion process takes care of much of the transformation from using C strings to NSStrings for you, but there are still places where you need to perform some of the conversion by hand. In most of these cases, you must change the type of the variable that captures a return value. The following table lists places in the Application Kit where a return value has changed to NSString, and you will need to change your variable's type.

Application Kit Classes	**Items That Are Now NSStrings**
TableHeadRule.eps ¬	
Button, Cell, Menu	command key equivalents

TableRule.eps ¬

NXBrowser TableRule.eps ¬	path separators

Other parts of the Application Kit now use NSArrays of NSStrings where they used to use arrays of C strings. The following table lists these areas. Again, you may need to change the declaration of a variable in your code accordingly.

Application Kit Classes TableHeadRule.eps ¬	**Items That Are Now NSArrays of NSStrings**
Application TableRule.eps ¬	services types
NXImage, NXImageRep TableRule.eps ¬	image file types
NXWorkspaceRequest Protocol TableRule.eps ¬	removable media
OpenPanel TableRule.eps ¬	open panel types
Pasteboard TableRule.eps ¬	pasteboard types
Printer TableRule.eps ¬	printer types
View, Window TableRule.eps ¬	dragged types

Pasteboard Types

Pasteboard's data types are now NSStrings instead of NXAtoms. To compare pasteboard data types, use the **isEqualToString:** method instead of the C == operator. The conversion process will flag the relevant code for you, and you should change it as shown in the following example.

Old Code

```
const NXAtom *myTypes = [aPasteboard types];
if (*myTypes[0] == NXFontPboardType) ...
```

New Code

```
NSArray *myTypes = [aPasteboard types];
if ([[myTypes objectAtIndex:0]
     isEqualToString:NSFontPboardType]) ...
```

The Empty String

The NSString constant **@""** is the empty string. Although NULL is a valid C string, **nil** is not a valid NSString. Methods that return NSStrings return an empty string in place of NULL. You also should use the empty string where there is no NSString.

ButtonCell's **stringValue** method used to return **""** if the state was 1 and NULL otherwise. Because Application Kit methods cannot return **nil** for NSString objects, this has changed as shown in the following table, and you must change code that tests the return value of **stringValue**.

Current Code	**Replace With:**
TableHeadRule.eps ¬	
strcmp([myButton stringValue], "")	[[myButton stringValue] isEqualToString:@"1"]
TableRule.eps ¬	

strcmp([myButton stringValue], NULL) [[myButton stringValue] isEqualToString:@"0"]
TableRule.eps ¬

NXCType.h Functions

NEXTSTEP Release 3 provided functions (for example, **NXToLower()**, **NXIsSpace()**, and **NXIsAlNum()**) that are similar to the ANSI C functions in the header file **ctype.h**. The NEXTSTEP versions of these functions are declared in the header file **NXCType.h**. The difference between the NXCType functions and their ANSI counterparts is that the NXCType functions take a NEXTSTEP-encoded character as input. The ANSI functions work only on ASCII characters.

The NXCType functions are not in OpenStep. The required string conversion changes them to their ASCII equivalents as shown below.

Old Code

```
char x = 'A';
char y;

if (NXIsUpper(x))
     y = NXToLower(x);
```

New Code

```
char x = 'A';
char y;

if (isupper(x))
     y = tolower(x);
```

A Foundation class, NSCharacterSet, defines character sets analogous to many of the functions in **ctype.h**, in addition to providing for the definition of customized sets. If you need to test non-ASCII characters or if you intend to localize the application you are converting, you should use NSCharacterSet instead of the **ctype.h** functions. After all six required conversion stages are complete, run the conversion script **StringConversion2.tops**, which helps you make this change. For more information, see the section ªOptional String Conversion.º

Service Providers

Service providers must now use NSStrings in the method that implements the service. You must convert these methods by hand. If your application provides a service, the method that performs that service should be declared like this:

```
-€serviceMethod:(NSPasteboard€*)pasteboard userData:(NSString€*)userData
€€€€€€€€€€€€€€€error:(NSString€**)errorString;
```

Note: For more information, see the on-line document **Services.rtf** under **/NextLibrary/Documentation/TasksAndConcepts/ProgrammingTopics**.

String Conversion Gotchas

After the required string conversion, look out for the following:

Formatted Strings

The conversion process sometimes gives the incorrect formatting argument for NSString objects and C strings. Sometimes an NSString is provided for a **%s** argument or a C string is provided for a **%@** (NSString) argument, as shown below. You must either change the control character or change the parameter.

Bad New Code

```
NSRunAlertPanel(...@"%s"...,
     [NSString stringWithCString:aString]...);
NSRunAlertPanel(...@"%@"..., [stringObject cString]...);
```

Good New Code

```
NSRunAlertPanel(...@"%s"..., aString...);
NSRunAlertPanel(...@"%@"..., stringObject...);
```

Specifying File Extensions

Previously, when you specified a file extension as an argument to a method such as **pathForResource:extension:**, you could begin the extension with the dot. When you specify a file extension in OpenStep, it is expected that you won't include the dot. The conversion process does not find these for you.

Old Code

```
[NXBundle pathForResource:"Employee" ofType:".eomodel"]
```

New Code

```
[NSBundle pathForResource:@"Employee" ofType:@"eomodel"]
```

Optional String Conversion

Optional

Perform the optional string conversion (**StringConversion2.tops**) to eliminate occurrences of NSString's **cString** and **stringWithCString:** methods in your code. The required string conversion put these methods in your code.

In many cases, the C strings supplied as arguments to the **stringWithCString:** methods can be converted to NSStrings, eliminating the need for these two messages.

The usefulness of this script varies; run it on a copy of your code first to see if it will help you. If you perform a lot of character manipulation, this script requires that you manually rewrite much of the code that performs character manipulation. If this is the case, and your application will not be localized, you may want to skip this conversion. If you do localize your application, even if you perform a lot of character manipulation, this script will help you because it increases your use of NSStrings.

Note: To learn how to run a single Tops script, see the on-line release note **/NextLibrary/Documentation/NextDev/ReleaseNotes/ConvertingYourCode.rtf**. Don't run this script until all six conversion stages are complete.

Character Manipulation

After conversion, you may have to rewrite code that accesses individual characters in a string. The conversion script assumes you are not trying to perform character manipulation on a C string, and it does not distinguish between a C string and a character pointer. For example, if you have a method like the following to extract the extension from a filename, it will be converted as shown below.

Old Code

```
char *extension = rindex(filename,'.');
if (extension) {
     *extension = 0;
     extension++;
}
```

Bad New Code

```
NSString *extension = [NSString stringWithCString:
     rindex(filename,'.')];
if (extension) {
```

```
    *extension = [NSString stringWithCString:0];
    [extension cString]++;
}
```

See the NSString class cluster description in the *Foundation Framework Reference* for guidance on converting code that accesses individual characters in a string. To solve this particular problem, NSString provides many methods that let you manipulate UNIX pathnames. You can extract the extension from a filename with one message, as shown below.

Good New Code

```
NSString *extension = [[NSString stringWithCString:filename]
            pathExtension];
```

The next example shows a more general case of accessing characters in a string. (It removes all occurrences of the letter `o' from a string.) To access individual characters in a C string, you use an array index. Accessing characters by index in an NSStrings is not desirable because it is slow and the string may contain non-ASCII characters that make sense only in a sequence. For example, the Unicode representation of ªÝº may be two characters long, ªeº and ªÂº. Instead, it is better to use ranges to locate characters, as shown below.

Old Code

```
char *aString;
int to, from;
int length = strlen(aString);

for (to = from = 0; from < length; from++) {
    if (aString[from] != 'o') {
     aString[to] = aString[from];
```

```
      to++;
     }
  }
  if (to < (length - 1)) aString[to] = '\0';
```

New Code

```
  NSMutableString *aString;
  NSRange rangeOfO;
  ...
  rangeOfO = [aString rangeOfString:@"o"];
  while (rangeOfO.length) {
       [aString deleteCharactersInRange:rangeOfO];
       rangeOfO = [aString rangeOfString:@"o"];
  }
```

Converting ctype.h Functions

During the required string conversion, the functions defined in **NXCType.h** (for example, **NXToLower()**, **NXIsSpace()**, and **NSIsAlNum()**) are converted to their ANSI C equivalents (defined in **ctype.h**). Mostly, these functions return whether a character has a particular property (is uppercase or lowercase, and so on). The ANSI C functions work on ASCII characters, whereas their NEXTSTEP counterparts work on characters in the NEXTSTEP encoding.

OpenStep also has the ability to handle such operations. It has a class named NSCharacterSet (in the Foundation Framework) that defines character sets. For example, the set of uppercase characters is a character set. To find out if a character has a particular property, you determine if it is a member of the appropriate character set with the method **characterIsMember:**. Some of the functions in **ctype.h** convert a character. In OpenStep, you use NSString methods to perform such operations.

The optional string conversion script flags all occurrences of a **ctype.h** function in your code with a warning. If you are going to internationalize the character or string in question, you should change the function into a **characterIsMember:** message sent to the NSCharacterSet mentioned in the warning message (or to the NSString method specified in the message). The following code shows an example.

Old Code

```
char x = 'Z';
char *y;
int i;
...
i = strlen(y) - 1;
if (isupper(x))
     y[i] = tolower(x);
```

New Code

```
unichar x = (unichar)'Z';
NSMutableString *y;
...
if ([[NSCharacterSet uppercaseLetterCharacterSet]
     characterIsMember:x])
     [y appendString:[[NSString stringWithCharacters:&x length:1]
         lowercaseString]];
```

Converting for Complete Localizability

One of the goals of OpenStep is to make it easier to write international applications. If you use NSString objects for all external strings (that is, strings that the user will see), you won't have to worry about the string's encoding when you are programming. In addition to NSString, OpenStep provides these classes and functions to help with localization. If you're converting a localized application, you'll want to use these classes and functions.

Even if you're not converting a localized application, these classes and functions may improve the portability of your code. If you want your application to run on the Microsoft Windows® platform, using NSStrings for such items window titles and file names can protect you from having to worry about the different character encodings used on Mach and Microsoft Windows.

NSDate

NSDate is a class cluster that stores dates in a localizable format. Use it in place of C functions such as those in the following table.

Function TableHeadRule.eps ¬	**NSDate Method**
time TableRule.eps ¬	date
ctime, strftime TableRule.eps ¬	descriptionWithCalendarFormat:timeZone:locale:

NSLog()

NSLog() is a function defined in the Foundation Framework that writes a formatted string to standard error. Replace all uses of **syslog()** in your code with **NSLog()**.

NSScanner

NSScanner scans an NSString for data. If you're using **sscanf()** or **fscanf()** to scan localized strings, you should replace those functions with NSScanner methods. See the description of NSScanner in the *Foundation Framework Reference* for more information.

NSFileManager

NSFileManager is an extension to the OpenStep API. It performs file manipulations for you, and it converts localized strings to a format acceptable to the file system. Use it in place of C functions such as those in the following table.

Function **TableHeadRule.eps ¬**	**NSFileManager Method**
getwd, getcwd TableRule.eps ¬	currentDirectoryPath
chown, chmod TableRule.eps ¬	changeFileAttributes:atPath:
stat, lstat TableRule.eps ¬	fileAttributesAtPath:traverseLink:
statfs TableRule.eps ¬	fileSystemAttributesAtPath:
readLink TableRule.eps ¬	pathContentOfSymbolicLinkAtPath:pathContent:
symlink TableRule.eps ¬	createSymbolicLinkAtPath:pathContent:
mkdir TableRule.eps ¬	createDirectoryAtPath:attributes:
creat, umask TableRule.eps ¬	createFileAtPath:contents:attributes:
fopen	contentsAtPath:

TableRule.eps ¬

chdir	changeCurrentDirectoryPath:

TableRule.eps ¬

link	linkPath:toPath:handler:

TableRule.eps ¬

rmdir, unlink	removeFileAtPath:handler:

TableRule.eps ¬

rename	movePath:toPath:handler:

TableRule.eps ¬