1 *Living in a Hybrid World*

The Enterprise Objects Framework documentation occasionally refers to a ªroot class.º This phrase does not lack precision, but reflects a current reality: NEXTSTEP has (temporarily) dual class hierarchies.

The version of NEXTSTEP that comes with the first release of the Enterprise Objects Framework has two root classes, and therefore it has two class hierarchies. There's Object, the root class that most NEXTSTEP developers are familiar with. And now there's also NSObject.

This document explains the new root class, discusses the new object management scheme supported by Foundation, and then concludes with a discussion of archiving, both of NSObjects and of mixed graphs containing objects that inherit from both NSObject and Object.

The New Root Class

NSObject is the root class for the Foundation Kit. and will soon become the root class for all NEXTSTEP classes. The Application Kit and most other NEXTSTEP classes are being converted to the class hierarchy rooted by NSObject.

The Foundation Kit (or, simply, Foundation) is a group of classes that replaces the Common classes (List, Hash, Storage, and so on). Foundation is more, however. As the name suggests, it lays down a foundation of object functionality that supports all other classes in NEXTSTEP. It provides base classes for things like strings, values, collections and storage. But more importantly, Foundation improves the persistence and distribution of objects within an object system that is independent of particular operating systems. Foundation also introduces paradigms and mechanisms that enrich the object-oriented development process, especially a new way to deallocate objects.

NSObjects and Objects are not interchangeable. In general, use NSObjects for very abstract objects or engines, or for things related to the Enterprise Objects Framework. Otherwise, use Objects. In particular, subclass Object for those objects that are related to kit or Interface Builder objects.

For all new code that you write using objects that inherit either from Object or NSObject, you should use the methods and object-management techniques discussed in the following sections. To make this possible, Foundation supplies a set of compatibility methods as a category on Object to make Objects behave like NSObjects (note, however, that the NSObject class has many additional methods, so reverse-compatibility is not guaranteed). The methods supplied by this category are:

+ allocWithZone
+ instanceMethodForSelector:
+ instancesRespondToSelector:
+ poseAsClass:
- autorelease
- conformsToProtocol:
- copyWithZone:
- dealloc
- doesNotRecognizeSelector
- isKindOfClass:

- isMemberOfClass:
- methodForSelector:
- perform:withObject:
- perform:withObject:withObject:
- release
- respondsToSelector:
- retain
- retainCount

The introduction to the Foundation Kit (found online, in **/NextLibrary/Documentation/NextDev/Foundation/IntroFoundation.rtfd**) contains a complete discussion of object management with NSObjects. When writing applications that contain both Objects and NSObjects, the primary thing to keep in mind is that *you should never send a **free** message to an NSObject; send **release** instead*. Although you can still send **free** to objects that inherit from Object, for compatibility with future releases of NEXTSTEP you should instead send **release** to these objects as well.

Archiving Objects

NSObject introduces a new mechanism for archiving and unarchiving objects that replaces the typed-stream approach of Object. This mechanism, implemented by the NSCoder, NSArchiver, and NSUnarchiver classes and the NSCoding protocol, encodes the objects of an application in a way that enhances their persistency and distributability. The repository of this encoded object information can be a file or an NSData object. You should archive any instance variables or other data critical to an object's state.

NSObject adopts the NSCoding protocol and so, by inheritance, all of its subclasses adopt it as well. Instances of

these subclasses receive, at the appropriate times in their life cycles, a message requesting that they encode themselves and a message asking that they decode and initialize themselves. You implement two NSCoding methods to intercept these messages: **encodeWithCoder:** and **initWithCoder:**.

Your implementation of these methods is similar to Object's **write:** and **read:** methods, but there are significant differences too. Both **encodeWithCoder:** and **initWithCoder:** should begin by invoking the corresponding superclass method. The invocation of **super**'s **initWithCoder:** returns the partially initialized object (**self**). End **initWithCoder:** by returning **self**, but do not return in **encodeWithCoder:**.

```
NSObject *myObject;    /* Assume this exists. */
id cell;
id view;
const char *flags;

- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
      myObject = [[coder decodeObject] retain];
    [coder decodeValuesOfObjCTypes:"@@s", &cell, &view, &flags];

    return self;
}

- (void)encodeWithCoder:(NSCoder *)coder
{

    [super encodeWithCoder:coder];
    [coder encodeObject:myObject];
    [coder encodeValuesOfObjCTypes:"@@s", &cell, &view, &flags];
}
```

NSCoder defines matching sets of methods for encoding and decoding objects of different types. In the above

example, **encodeValuesOfObjCTypes:** takes a format string consisting of the same type specifiers used by **NXWriteTypes()** (used in **read:**). Following this is a variable sequence of arguments, each of which is the address of a variable (usually instance variable). Note that the data, by type, must be decoded in the same sequence as it was encoded.

Just as **NXWriteRootObject()** initiates archiving in classes that inherit from Object by invoking **write:**, NSArchiver's **archiveRootObject:toFile:** initiates archiving in the NSObject world, invoking **encodeWithCoder:**. NSUnarchiver's **unarchiveObjectWithFile:** initiates unarchiving in NSObject instances by invoking **initWithCoder:** in them. Never invoke **encodeWithCoder:** or **initWithCoder:** directly.

Archiving Mixed Object Graphs

In the dual root-class situation, you might have a class that inherits from one root class but that has some instance variables that inherit from the other root class. When it comes to archiving objects that inherit from Object along with objects that inherit from NSObject, there might seem to be a problem: Do you use the old approach to archiving (**write:**), or the new (**encodeWithCoder:**)?

As an example, consider these declarations:

```
@interface Author:NSObject
{
    NSString *authorID;        // These inherit from NSObject
    NSString *firstName;
    NSString *lastName;
    NSString *address;
    NSString *city;
    NSString *state;
    int contract;        // Scalar type
```

```
    List *titles;        // Inherits from Object
}
```

The Author class inherits from NSObject, but it has a List object (**titles**) as one of its instance variables. List inherits from the Object class, and List objects are thus supposed to be archived with the **NXWriteObject()** (or similar) function within the **write:** method.

This mix of archived object hierarchies makes compatibility with future releases of NEXTSTEP a problem. To get around this problem, NEXTSTEP provides some compatibility methods and functions for you to use when archiving and unarchiving in these situations.

Archiving NSObjects That Contain Objects

If you have an NSObject subclass with objects in its 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 examples:

```
- (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;
}
```

Archiving Objects That Contain NSObjects

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.

Assuming the previous declarations of instance variables are now made for a class that inherits from Object, the archiving and unarchiving methods would look like this example.

```
- 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:

1. 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, NEXTSTEP keeps enough information about the

objects so that the cycles are detected and objects that are pointed to by many other objects are only archived 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.

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

Forward Compatibility

The techniques for archiving graphs of mixed objects described above are the only ones guaranteed to compatible with later releases of NEXTSTEP. Pre-existing archives that consist of objects which all inherit from Object, and new archives consisting entirely of objects that inherit from NSObject will also be compatible with future releases of NEXTSTEP. If you add **read:** and **write:** methods to NSObject (and its descendants) your archives are guaranteed to be *incompatible* with future versions of NEXTSTEP.