NXApp, Summer 1994. Volume 1, Issue 3.

Sneak Preview: The New Foundation Kit

written by **NeXT Developer Publications**

The Foundation Kit is a new set of Objective C classes that provides useful primitive
object classes, and introduces several paradigms to avoid confusion in common situations and introduce consistency across class hierarchies.

INTRODUCTION

The Foundation Kit defines a base layer of Objective C classes for OpenStep™. In addition to providing useful primitive object classes, it introduces paradigms that define functionality not covered by the Objective C language. The Foundation Kit is designed with these goals in mind:

- Provide a small set of basic utility classes
- Make software development easier by introducing consistent conventions for things such as deallocation
- Support Unicode strings, object persistence, and object distribution
- Provide a level of operating system independence, to ensure portability

The Foundation Kit includes the root object class; classes representing basic data types such as strings and byte arrays, and collections of other objects; and classes representing system information such as dates and communication ports between applications.

The Foundation Kit introduces several paradigms to avoid confusion in common situations and to introduce a level of consistency across class hierarchies. This is done with some standard policies, such as that for object ownership (that is, who's responsible for disposing of objects), and with Objective C protocols like NSEnumerator. These new paradigms reduce

the number of special and exceptional cases in the API, and allow you to code more efficiently by reusing the same mechanisms with various kinds of objects.

Some of these design goals (especially the last one) are longer-term and haven't been completely achieved in the first release of Foundation Kit. For example, although applications can work with Unicode strings, there's no support for displaying text in Unicode's wide range of characters or for getting Unicode input from the user.

FOUNDATION KIT CLASSES AND PROTOCOLS

The OpenStep class hierarchy is rooted in the Foundation Kit's NSObject class (see Figure 7). The remainder of the Foundation Kit consists of several related groups of classes as well as a few individuals. Most of the groups form what are called *class clusters*Ðabstract classes that work as umbrella interfaces to a versatile set of private subclasses. NSString and NSMutableString, for example, act as brokers for instances of various private subclasses optimized for different kinds of storage needs. Depending on the method you use to create a string, an instance of the appropriate optimized class will be returned to you. See ªClass Clustersº for a full treatment of this concept.

F0.eps ,

Figure 7: *The Foundation Kit class inheritance hierarchy*

The first group of classes handles object storage. They hold arrays of objects or bytes, or store objects by key. The NSValue and NSNumber classes allow you to store arrays of simple C data values in an NSArray or other storage object.

The next group of classes represents text strings and characters. The NSCharacterSet classes represent various groupings of characters in the NSString and NSScanner classes. The NSString classes represent text strings and provide methods for searching, combining, and comparing strings. An NSScanner is used to scan numbers and words from an NSString object.

NSAutoreleasePools are used to implement the delayed-release feature of the Foundation Kit, as described in ªObject Ownership and Automatic Disposal.º

The NSDate and NSTimeZone classes store times and dates. They offer methods for calculating dates and time differences, for displaying dates and times in many formats, and for adjusting times and dates based on location in the world.

The Foundation Kit defines only a few protocols, the most significant of which is the NSObject
protocol. It declares the methods that any object must implement to be considered a ªfirst-classº object. The other protocols include NSCopying and NSMutableCopying, which declare methods for making copies of objects. The two separate protocols allow for making immutable or mutable copies of objects that may or may not be mutable.

Final determination of which classes are public and private hasn't yet been made. You may see header files for other classes in the Foundation Kit header file directory and some classes listed here may not appear in the header files.

OBJECT OWNERSHIP AND AUTOMATIC DISPOSAL

Note As a preface to this discussion, note that in Foundation you implement a **dealloc** method instead of a **free** method. **dealloc** is always invoked indirectly through **release**, which you use in place of **free**. So, you speak of ªreleasingº an object instead of ªfreeingº it. These terms and
methods will be introduced in a new Objective C document.

In an Objective C program objects are constantly creating and disposing of other objects. Much
of the time an object creates things for private use and can dispose of them as it needs. However, when an object passes something to another object through a method invocation, the lines of ownershipÐand responsibility for disposalÐblur. Suppose, for example, that you have a Gadget object that contains a number of Sprocket objects, which another object accesses with this method:

```
± (NSArray *)sprockets
```

This declaration says nothing about who should release the returned array. If the Gadget object returned an instance variable, it's responsible; if the Gadget created an array and returned it, the recipient is responsible. This problem applies both to objects returned by a method and objects passed in as arguments to a method.

Ideally a body of code should never be concerned with releasing something it didn't create. The Foundation Kit therefore sets this policy: *If you create an object you alone are responsible for releasing it.* If you didn't create the object, you don't own it and shouldn't release it.

When you write a method that creates and returns an object, then, that method is responsible
for releasing the object. It's clearly not fruitful to dispose of an object before the recipient of the object gets it, however. What's needed is a way to mark an object for later release, so that it will be properly disposed of after the recipient has had a chance to use it. The Foundation Kit provides just such a method.

Marking objects for disposal

The **autorelease** method, defined by NSObject, marks the receiver for later release. By autoreleasing an objectÐthat is, by sending it an **autorelease** messageÐyou declare that you don't need the object to exist beyond the scope you sent **autorelease** in. When your code completely finishes executing and control returns to the application object (that is, at the end of the event loop), the application object releases the object. The **sprockets** methods above could be implemented in this way:

```
± (NSArray *)sprockets
{
   NSArray *array;

   array = [[NSArray alloc] initWithObjects:mainSprocket,
                       auxiliarySprocket, nil];
   return [array autorelease];
}
```

When another method gets the array of Sprockets, that method can assume that the array will be disposed of when it's no longer needed, but can still be safely used anywhere within its scope (with certain exceptions; see ªValidity of Shared Objectsº). It can even return the array to its invoker, since the application object defines the bottom of the call stack for your code. The **autorelease** method thus allows every object to use other objects without worrying about disposing of them.

Note Just as it's an error to release an object after it's already been deallocated, it's an error to send so many **autorelease** messages that the object would later be released after it had already been deallocated. You should send **release** or **autorelease** to an object only as many times as are allowed by its creation (one) plus the number of **retain** messages *you* have sent it (**retain** messages are described in the next section).

Retaining objects

There are times when you don't want a received object to be disposed of; for example, you may need to cache the object in an instance variable. In this case, only you know when the object is no longer needed, so you need the power to ensure that the object is not disposed of while you are still using it. You do this with the **retain** method, which stays the effect of a pending **autorelease** (or preempts a later **release** or **autorelease** message). By retaining an object you ensure that it won't be deallocated until you're done with it. For example, if your object allows its main Sprocket to be set, you might want to retain that Sprocket like this:

```
± (void)setMainSprocket:(Sprocket *)newSprocket
{
   [mainSprocket autorelease];
   mainSprocket = [newSprocket retain]; /* Claim the new Sprocket. */
   return;
}
```

Now, **setMainSprocket:** might get invoked with a Sprocket that the invoker intends to keep around, which means your object would be sharing the Sprocket with that other object. If that object changes the Sprocket, your object's main Sprocket changes. You might want that, but if your Gadget needs to have its own Sprocket the method should make a private copy:

```
± (void)setMainSprocket:(Sprocket *)newSprocket
{
   [mainSprocket autorelease];
   mainSprocket = [newSprocket copy]; /* Get a private copy. */
   return;
}
```

Note that both of these methods autorelease the original main Sprocket, so they don't need to check that the original main Sprocket and the new one are the same. If they simply

released the original when it was the same as the new one, that Sprocket would be released and possibly deallocated, causing an error as soon as it was retained or copied. Although they could store the old main Sprocket and release it later, that kind of code tends to be slightly more complex. For example:

```
± (void)setMainSprocket:(Sprocket *)newSprocket
{
   Sprocket *oldSprocket = mainSprocket;
   mainSprocket = [newSprocket copy];
   [oldSprocket release];
   return;
}
```

Validity of shared objects

The Foundation Kit's ownership policy limits itself to the question of when you have to dispose of an object; it doesn't specify that any object received in a method *must* remain valid throughout that method's scope. A received object nearly always becomes invalid when its owner is released, and usually becomes invalid when its owner reassigns the instance variable holding that object. Any method other than **release** that immediately disposes of an object is documented as doing so.

For example, if you ask for an object's main Sprocket and release the object, you have to consider the main Sprocket gone, because it belonged to the object. Similarly, if you ask for the main Sprocket and then send **setMainSprocket:** you can't assume that the Sprocket you received remains valid:

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [myObject mainSprocket];

/* If this releases the original Sprocket... */
[myObject setMainSprocket:newMainSprocket];

/* ...then this causes the application to crash. */
[oldMainSprocket anyMessage];
```

setMainSprocket: may release the object's original main Sprocket, possibly rendering it

invalid. Sending any message to the invalid Sprocket would then cause your application to crash. If you need to use an object after disposing of its owner or rendering it invalid by some other means, you can retain and autorelease it before sending the message that would invalidate it:

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [[[myObject mainSprocket] retain] autorelease];
[myObject setMainSprocket:newMainSprocket];
[oldMainSprocket anyMessage];
```

Retaining and autoreleasing **oldMainSprocket** guarantees that it will remain valid throughout your scope, even though its owner may release it when you send **setMainSprocket:**.

Summary
Now that the concepts behind the Foundation Kit's object ownership policy have been introduced, they can be expressed as a short list of rules:

- If you allocate, copy, or retain an object, you are responsible for releasing the newly created object with **release** or **autorelease**. Any other time you receive an object, you're not responsible for releasing it.
- A received object is normally guaranteed to remain valid within the method it was received in. That method may also safely return the object to its invoker.
- If you need to store a received object in an instance variable, you must retain or copy it.
- Use **retain** and **autorelease** when needed to prevent an object from being invalidated as a normal side-effect of a message.

CLASS CLUSTERS
The Foundation Kit's architecture makes extensive use of class clusters. Class clusters group a number of private, concrete subclasses under a public, abstract superclass. The grouping of classes in this way simplifies the publicly visible architecture of an object-oriented kit without reducing its functional richness.

Simple concept, complex interface

To illustrate the class cluster architecture and its benefits, consider the problem of constructing a class hierarchy that defines objects to store numbers of different types (**char**s, **int**s, **float**s, **double**s). Since numbers of different types have many features in common (they can be converted from one type to another and can be represented as strings, for example), they could be represented by a single class. However, their storage requirements differ, so it's inefficient to represent them all by the same class. This suggests the architecture shown in Figure 8.

F2_SimpleHeirarchy.eps ¬

Figure 8: *A simple hierarchy for number classes*

Number is the abstract superclass that declares in its methods the operations common to its subclasses. However, it doesn't declare an instance variable to store a number. The subclasses declare such instance variables and share in the programmatic interface declared by Number.

So far, this design is relatively simple. However, if the commonly used modifications of these basic C types are taken into account, the diagram looks more like the one in Figure 9.

F3_CompleteHeirarchy.eps ¬

Figure 9: *A more complete number class hierarchy*

The simple conceptÐcreating a class to hold number valuesÐcan easily burgeon to over a dozen classes. The class cluster architecture presents a design that reflects the simplicity of the concept.

Simple concept, simple interface

Applying the class cluster design to this problem yields the hierarchy shown in Figure 10 (private classes are in gray).

881924_F4_ClassCluster.eps ¬

Figure 10: *Class cluster architecture applied to number classes*

Users of this hierarchy see only one public class, Number, so how is it possible to allocate instances of the proper subclass? The answer is in the way the abstract superclass handles instantiation.

Creating instances

The abstract superclass in a class cluster must declare methods for creating instances of its private subclasses. It's the superclass's responsibility to dispense an object of the proper subclass based on the creation method that you invokeÐyou don't, and can't, choose the class of the instance.

In the Foundation Kit, you generally create an object by invoking a **+** *className...* method or the **alloc...** and **init...** methods. Taking the Foundation Kit's NSNumber class as an example, you could send these messages to create number objects:

```
NSNumber *aChar = [NSNumber numberWithChar:'a'];
NSNumber *anInt = [NSNumber numberWithInt:1];
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

(This style of instantiation creates objects that will be deallocated automaticallyÐsee ªObject Ownership and Automatic Disposalº earlier in this article for more information. Many classes also provide the standard **alloc...** and **init...** methods to create objects that require you to manage their deallocation.)

Each object returnedÐ*aChar*, *anInt*, *aFloat*, and *aDouble*Ðmay belong to a different private subclass (and in fact does). Although each object's class membership is hidden, its interface is public, being the interface declared by the abstract superclass, NSNumber.

You could consider the *aChar*, *anInt*, *aFloat*, and *aDouble* objects to be instances of the NSNumber class, since they're created by NSNumber class methods and accessed through instance methods declared by NSNumber. However, this isn't precisely correct, as explained above, so this documentation uses a shorthandÐa lowercase version of the class nameÐto refer to such objects. Thus, the *aChar*, *anInt*, *aFloat*, and *aDouble* objects are called *number objects*.

Class clusters with multiple public superclasses

In the example above, one abstract public class declares the interface for multiple private subclasses. This is a class cluster in the purest sense. It's also possible, and often desirable, to have two (or possibly more) abstract public classes that declare the interface for the cluster. This is evident in the Foundation Kit, which includes the clusters listed in Figure 11.

Class cluster	Public superclasses	Name of instance
NSData	NSData	data object
	NSMutableData	mutable data object
NSArray	NSArray	array object
	NSMutableArray	mutable array object
NSDictionary	NSDictionary	dictionary object
	NSMutableDictionary	mutable dictionary object
NSString	NSString	string object
	NSMutableString	mutable string object
NSValue	NSValue	value object
	NSNumber	number object

Figure 11: *Abstract public classes for class clusters*

Other clusters of this type also exist, but these clearly illustrate how two abstract nodes cooperate in declaring the programmatic interface to a class cluster. In each of these clusters, one public node declares methods that all cluster objects can respond to, and the other node declares methods that are appropriate only for cluster objects that allow their contents to be modified.

This factoring of the cluster's interface helps make an object-oriented kit's programmatic interface more expressive. For example, imagine a Book object that declares this method:

```
- (NSString *)title;
```

The Book object could return its own instance variable or create a new string object and return thatÐit doesn't matter. It's clear from this declaration that the returned string can't be modified. Any attempt to modify the returned object will elicit a compiler warning.

Creating subclasses within a class cluster

The class cluster architecture involves a trade-off between simplicity and extensibility: Having a few public classes stand in for a multitude of private ones makes it easier to learn

and use the classes in a kit but somewhat harder to create subclasses within any of the clusters. However, if it's rarely necessary to create a subclass, then the cluster architecture is clearly beneficial. Clusters are used in the Foundation Kit in just these situations.

If you find that a cluster doesn't provide the functionality your program needs, then a subclass may be in order. For example, imagine that you want to create an array object whose storage is file-based rather than memory-based as in the NSArray class cluster. Since you are changing the underlying storage mechanism of the class, you'd have to create a subclass.

On the other hand, in some cases it might be sufficient (and easier) to define a class that embeds within it an object from the cluster. Let's say that your program needs to be alerted whenever some data is modified. In this case, creating a simple cover for a data object that the Foundation Kit defines may be the best approach. An object of this class could intervene in messages that modify the data, intercepting the messages, acting on them, and then forwarding them to the embedded data object.

In summary, if you need to manage your object's storage, create a true subclass. Otherwise, create a composite object, one that embeds a standard Foundation Kit object in an object of your own design. The sections below give more detail on these two approaches.

A true subclass

A new class that you create within a class cluster must:

- Be a subclass of the cluster's abstract superclass
- Declare its own storage
- Override the superclass's primitive methods

Since the cluster's abstract superclass is the only publicly visible node in the cluster's hierarchy, the first point is obvious. This implies that the new subclass will inherit the cluster's interface but no instance variables, since the abstract superclass declares none. Thus the second point: The subclass must declare any instance variables it needs. Finally, the subclass must override any method it inherits that directly accesses an object's instance variables. Such methods are called *primitive methods*.

A class's primitive methods form the basis for its interface. For example, take the NSArray

class, which declares the interface to objects that manage arrays of objects. In concept, an array stores a number of data items, each of which is accessible by index. NSArray expresses this abstract notion through its two primitive methods, **count** and **objectAtIndex:**. With these methods as a base, other methodsÐ*derived methods*Ðcan be implemented, for example:

- **lastObject** Find the last object by sending the array object this message:
  [self objectAtIndex:[self count] ±1]
- **containsObject** Find an object by repeatedly sending the array object an **objectAtIndex:** message, each time incrementing the index until all objects in the array have been tested

The division of an interface between primitive and derived methods makes creating subclasses easier. Your subclass must override inherited primitives, but having done so can be sure that all derived methods that it inherits will operate properly.

The primitive-derived distinction applies to the interface of a fully initialized object. The question of how **init...** methods should be handled in a subclass also needs to be addressed.

In general, a cluster's abstract superclass declares a number of **init...** and **+** *className* methods. As described in ªCreating Instancesº earlier in this article, the abstract class decides which
concrete subclass to instantiate based on your choice of **init...** or **+** *className* method. You can consider that the abstract class declares these methods for the convenience of the subclass.
Since the abstract class has no instance variables, it has no need of initialization methods.

Your subclass should declare its own **init...** (if it needs to initialize its instance variables) and possibly **+** *className* methods. It should not rely on any of those that it inherits. To maintain its link in the initialization chain, it should invoke its superclass's designated initializer within its own designated initializer method. (See the *NEXTSTEP Object-Oriented Programming and the Objective C Language* manual for a discussion of the designated initializers.) Within a class
cluster, the designated initializer of the abstract superclass is always **init**.

True subclasses: an example

An example will help clarify the foregoing discussion. Let's say that you want to create a

subclass of NSArray, named MonthArray, that returns the name of a month given its index position. However, a MonthArray object won't actually store the array of month names as an instance variable. Instead, the method that returns a name given an index position (**objectAtIndex:**) will return constant strings. Thus, only 12 string objects will be allocated, no matter how many MonthArray objects exist in an application.

The MonthArray class is declared as:

```
#import <NSFoundationKit/NSFoundationKit.h>
@interface MonthArray : NSArray
{
}

+ array;
- (unsigned)count;
- objectAtIndex:(unsigned)index;

@end
```

Note that the MonthArray class doesn't declare an **init...** method since it has no instance variables to initialize. For convenience, it declares the **array** method so that users can easily create autoreleased instances. The **count** and **objectAtIndex:** methods simply cover the inherited primitive methods, as described above.

The implementation of the MonthArray class looks like this:

```
#import "MonthArray.h"

@implementation MonthArray

static NSString *Months[] = { @"January", @"February", @"March",
   @"April", @"May", @"June", @"July", @"August", @"September",
   @"October", @"November", @"December" };

+ array
{
   return [[[self alloc] init] autorelease];
}

- (unsigned)count
```

```
{
   return (sizeof(Months) / sizeof(Months[0]));
}

- objectAtIndex:(unsigned)index
{
   if (index < [self count]) {
      return Months[index];
   }
   NSRaise(NSRangeException, @"*** %s: index (%d) beyond bounds
      0 to %d)", sel_getName(_cmd), index, [self count] - 1);
   return nil; /* Should never get here */
}

@end
```

Since MonthArray overrides the inherited primitive methods, the derived methods that it inherits will work properly without being overridden. NSArray's **lastObject**, **containsObject:**, **sortedArrayUsingSelector:**, **objectEnumerator**, and other methods work without problems for MonthArray objects.

A composite object

By embedding a private cluster object in an object of your own design, you create a composite object. This composite object can rely on the cluster object for its basic functionality, intercepting only messages that it wants to handle in some particular way. Using this approach reduces the amount of code you must write and lets you take advantage of the tested code provided by the Foundation Kit.

Figure 12 shows how one might view a composite object:

F5_CompositeObj.eps ¬

Figure 12: *Embedding a cluster object*

The composite object must declare itself to be a subclass of the cluster's abstract node. As a subclass, it must override the superclass's primitive methods. It can also override derived methods, but this isn't necessary since the derived methods work through the primitive ones.

Using NSArray's **count** method as an example, the intervening object's implementation of a

method it overrides can be as simple as:

```
- (unsigned)count
{
   return [embeddedObject count];
}
```

However, your object could put code for its own purposes in the implementation of any method it overrides.

An intervening object: an example

To illustrate the use of an intervening object, imagine you want a mutable array object that tests changes against some validation criteria before allowing any modification to the array's contents. The example that follows describes a class called ValidatingArray, which contains a standard mutable array object. ValidatingArray overrides all of the primitive methods declared in its superclasses, NSArray and NSMutableArray. It also declares the **array**, **validatingArray**, and **init** methods, which can be used to create and initialize an instance:

```
#import <foundation/foundation.h>

@interface ValidatingArray : NSMutableArray
{
   NSMutableArray *embeddedArray;
}

+ array;
+ validatingArray;
- init;
- (unsigned)count;
- objectAtIndex:(unsigned)index;
- (void)addObject:object;
- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
- (void)removeLastObject;
- (void)insertObject:object atIndex:(unsigned)index;
- (void)removeObjectAtIndex:(unsigned)index;

@end
```

The implementation file shows how, in a ValidatingArray's **init** method, the embedded object

is created and assigned to the *embeddedArray* variable. Messages that simply access the array but don't modify its contents are relayed to the embedded object. Messages that could change the contents are scrutinized (here in pseudocode) and relayed only if they pass the hypothetical validation test.

```
#import "ValidatingArray.h"

@implementation ValidatingArray

- init
{
   embeddedArray = [[NSMutableArray array] retain];
   return self;
}



+ array
{
   return [self validatingArray];
}

+ validatingArray
{
   return [[[self alloc] init] autorelease];
}

- (unsigned)count
{
   return [embeddedArray count];
}

- objectAtIndex:(unsigned)index
{
   return [embeddedArray objectAtIndex:index];
}

- (void)addObject:object
{
   if (/* modification is valid */) {
      [embeddedArray addObject:object];
   }
```

```
}

- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
{
   if (/* modification is valid */) {
      [embeddedArray addObject:object];
   }
}
- (void)removeLastObject;
{
   if (/* modification is valid */) {
      [embeddedArray removeLastObject];
   }
}
- (void)insertObject:object atIndex:(unsigned)index;
{
   if (/* modification is valid */) {
      [embeddedArray insertObject:object atIndex:index];
   }
}
- (void)removeObjectAtIndex:(unsigned)index;
{
   if (/* modification is valid */) {
      [embeddedArray removeObjectAtIndex:index];

}
```

The Developer Publications group writes all of NeXT's developer-related publications and creates examples for NEXTSTEP kits and frameworks. In addition, the group works with the software engineers to help ensure high-quality, consistent, and clean APIs and user interfaces. They can be reached by e-mail at **DevPubs_feedback@next.com**; they welcome comments and suggestions on this and all other NEXTSTEP developer documentation.

Next Article NeXTanswer #1998 **A Methodology for Message-Based Undo and Animation**
Previous article NeXTanswer #2002 **An Introduction to the Enterprise Objects Framework**
Table of contents http://www.next.com/HotNews/Journal/NXapp/Summer1994/ContentsSummer1994.html