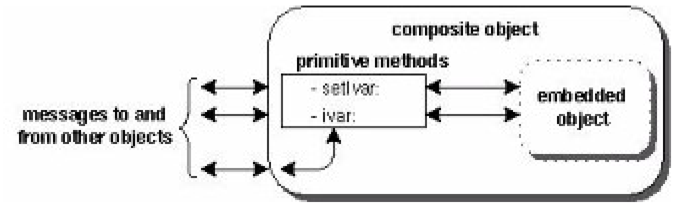
A Composite Object

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, only intercepting 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 Framework.

A composite object can be viewed in this way:



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.

A Composite Object: An Example

To illustrate the use of a composite 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;
}

+ 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 pseudo-code) and relayed only if they pass the hypothetical validation test.

```
#import "ValidatingArray.h"

@implementation ValidatingArray

- init
{
    embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
    return self;
}

+ 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 replaceObjectAtIndex:index withObject: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];
    }
}
```

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 (described below)

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 methods—*derived methods*—can be implemented, for example:

Derived Method	**Possible Implementation**
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 "Creating Instances" above, the abstract class decides which concrete subclass to instantiate based 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**.

Add-on Services

You typically define services when you create your application and advertise them in the **Info.plist** file of the application's bundle. The services facility also allows you to advertise services outside of the application bundle, enabling you to create "add-on" services after the fact. This is where the **NSUserData** entry becomes truly useful: You can define a single message in your application that performs actions based on the user data provided, such as running the user data string as a UNIX command (which the Terminal application does) or treating it as a special argument in addition to the selected data that gets sent through the pasteboard.

To define an add-on service, you create a bundle with a **.service** extension that contains an **Info.plist** file, which in turn contains the add-on service specification. You then put this bundle into a **Services** directory in the library search path (**~/Library**, **/LocalLibrary**, **/NextLibrary**). The services facility scans these directories when the user logs in and takes note of which services are defined; you can force this scanning by running the **make_services** UNIX command. If your application creates a service at run time and needs it to be available immediately, it calls this function to force scanning:

void **NSUpdateDynamicServices(**void**)**

Architectural Overview

You can think of the text-handling system as having three distinct layers of API. For most typical uses, the general-purpose programmatic interface of the NSTextView class is all you need to learn. If you need more flexible programmatic access to the text, you'll need to learn about the storage layer and the NSTextStorage class. And, of course, to access all the available features, you can learn about and interact with any of the classes that support the text-handling system. The following discussion presents these three layers.

Assembling the Text System by Hand

You build the network of objects that make up the text-handling system from the bottom up, starting with the NSTextStorage object. Here's the process:

1. **Set up an NSTextStorage object.**

   You create an NSTextStorage object in the normal way, using the **alloc** and **init...** messages. In the simplest case, where there's no initial contents for the NSTextStorage, the initialization looks like this

   ```
   textStorage = [[NSTextStorage alloc] init];
   ```

   If, on the other hand, you want to initialize an NSTextStorage object with rich text data from a file, the initialization looks like this (assume **fileName** is defined):

   ```
   NSAttributedString *attrString = [NSAttributedString
       attributedStringFromRTF:[NSData dataWithContentsOfFile:fileName]];


   textStorage = [[NSTextStorage alloc]
       initWithAttributedString:attrString];
   ```

   We've assumed that **textStorage** is an instance variable of the object that contains this method. When you create the text-handling system by hand, you need to keep a reference only to the NSTextStorage object as we've done here. The other objects of the system are owned either directly or indirectly by this NSTextStorage object, as you'll see in the next steps.

2. **Set up an NSLayoutManager object:**

   Next, create an NSLayoutManager object:

   ```
   NSLayoutManager *layoutManager;


   layoutManager = [[NSLayoutManager alloc] init];
   [textStorage addLayoutManager:layoutManager];
   [layoutManager release];
   ```

   Note that **layoutManager** is released after being added to **textStorage**. This is because the NSTextStorage object retains each NSLayoutManager that's added to it—that is, the NSTextStorage object *owns* its NSLayoutManagers.

   The NSLayoutManager needs a number of supporting objects—such as those that help it generate glyphs or position text within a text container—for its operation. It automatically creates these objects (or connects to existing ones) upon initialization. You only need to connect the NSLayoutManager to the NSTextStorage object and to the NSTextContainer object, as seen in the next step.

3. **Set up an NSTextContainer object.**

Next, create an NSTextContainer and initialize it with a size. Assume that **theWindow** is defined and represents the window that displays the text view.

```
NSRect cFrame = [[theWindow contentView] frame];

NSTextContainer *container;


container = [[NSTextContainer alloc]
    initWithContainerSize:cFrame.size];
[layoutManager addTextContainer:container];
[container release];
```

Once you've created the NSTextContainer, you add it to the list of containers that the NSLayoutManager owns, and then you release it. The NSLayoutManager now owns the NSTextContainer and is responsible for releasing it when it's no longer needed. If your application has multiple NSTextContainers, you can create them and add them at this time.

4. **Set up an NSTextView object.**

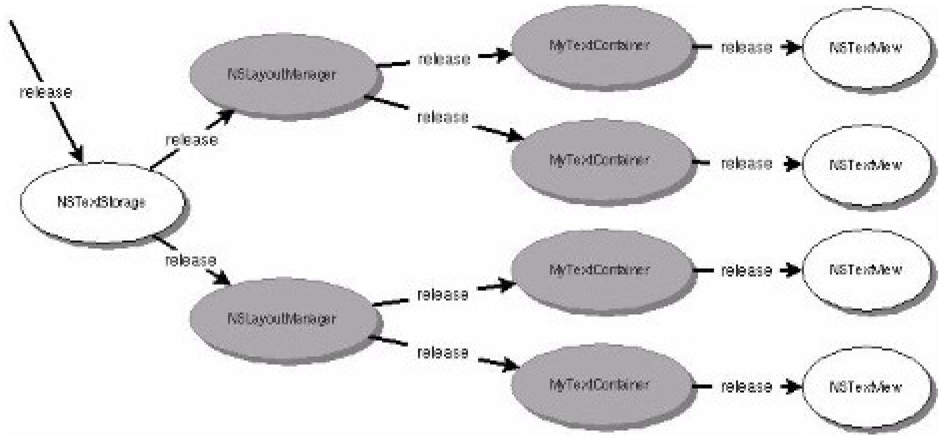
Finally, create the NSTextView (or NSTextViews) that displays the text:

```
NSTextView *textView = [[NSTextView alloc]
    initWithFrame:cFrame textContainer:container];


[theWindow setContentView:textView];
[theWindow makeKeyAndOrderFront:nil];


[textView release];
```

Note that we use **initWithFrame:textContainer:** to initialize the NSTextView. This initialization method does nothing more than what it says: initialize the receiver and set its text container. This is in contrast to **initWithFrame:**, as discussed in "Creating an NSTextView Programmatically," which not only initializes the receiver, but creates and interconnects the network of objects that make up the text-handling system. Once the NSTextView has been initialized, it's added to the window, which is then displayed. Finally, you release the NSTextView.

Note that in creating the text-handling network by hand, we created four objects but then released three as they were added to the network. We are left with a reference only to the NSTextStorage object. The NSTextView is retained by both its NSTextContainer and its superview, though; to fully destroy this group of text objects you must send **removeFromSuperview** to the NSTextView object and then release the NSTextStorage object.

An NSTextStorage object is conceptually the owner of any network of text-handling objects, no matter how complex. When you release the NSTextStorage object, it releases its NSLayoutManagers, which release their NSTextContainers, which in turn release their NSTextViews.



However, recall that the text system implements a simplified ownership policy for those whose only interaction with the system is through the NSTextView class. See "Creating an NSTextView Programmatically" for more information.

The code in the four steps above overlooks an important issue: resizing. As the window is resized, does the text rewrap within the new boundaries? What happens when there's more text than fits within the content view of the NSWindow? For information on these subjects, see "Putting an NSTextView Object in an NSScrollView."

Changing Character Attributes

Interface Builder’s Font and Text menus offer many standard commands for altering text attributes: Bold, Superscript, Center, and so on. These work by invoking standard action methods, such as **changeFont:**, **superscript:**, and **alignCenter:**, that effect a specific change in one step. But how do you define and implement new commands?

Let’s say that you want to define a command that emphasizes the selected text in some way. For example, using Interface Builder, you wish to add a menu command that sends an **emphasizeText:** message, perhaps to a custom object that owns an NSTextView. The custom object then sets the font in the NSTextView.

In doing so, the custom object can invoke any NSTextView method that changes attributes, but it must first ask for permission to do so and inform the NSTextView that changes are occurring so that the NSTextView can batch them together and send out the appropriate notifications to observers. Given this, and assuming the custom object has an instance variable (named **theTextView**) that identifies the NSTextView containing the selection, you can implement the **emphasizeText:** method like this:

```
- (void)emphasizeText:(id)sender
{
    NSRange changeCharRange = [theTextView
        rangeForUserCharacterAttributeChange];


    if ([theTextView shouldChangeTextInRange:changeCharRange
                     replacementString:nil]) {
        [theTextStore beginEditing];


        [myTextView setFont:[NSFont fontWithName:@"Helvetica-Oblique"
            size:12.0] range:changeCharRange];


        [theTextStore endEditing];
        [theTextView didChangeText];
    }

    return;
}
```

The custom object gets the range of the selected text and then applies a new font to that range. It then determines the range of characters that should be changed, and proceeds to attempt the change. To do so it invokes **shouldChangeTextInRange:replacementString:**, which gives the NSTextView’s delegate a chance to reject the change. If the change is approved, this method sets the font of the characters being changed, bracketing the change with **beginEditing** and **endEditing** messages that allow the NSTextView to optimize multiple changes (though only one change is made here). Finally, this method invokes **didChangeText** to send out the appropriate delegate message and notifications.

This implementation sent a **setFont:range:** message to the NSTextView to effect its change. NSTextView defines other, similar, methods to set some common attributes (such as font, text color, and alignment). These are “cover” methods that hide the work of invoking the NSTextStorage methods that actually modify the attributed string. If you want to set attributes other than those accessible through the NSTextView API, you have to interact more intimately with the NSTextStorage object.

Fortunately, working with NSTextStorage is quite straightforward. For example, a reimplemented **emphasizeText:** method that acts on the underlying NSTextStorage object looks like this:

```
- (void)emphasizeText:(id)sender
{
    NSTextStorage *theTextStore = [theTextView textStorage];
    NSRange changeCharRange = [theTextView
        rangeForUserCharacterAttributeChange];


    if (changeCharRange.location == NSNotFound) return;


    if ([theTextView shouldChangeTextInRange:changeCharRange
                     replacementString:nil]) {
        [theTextStore beginEditing];


        [theTextStore addAttribute:NSFontAttributeName
            value:[NSFont fontWithName:@"Helvetica-Oblique" size:12.0]
            range:changeCharRange];


        [theTextStore endEditing];
        [theTextView didChangeText];
    }

    return;
}
```

Except for interacting with the NSTextStorage instead of the NSTextView, this implementation is identical to the first one, asking for permission to make the change, and informing the NSTextView as things proceed.

Regarding the change itself: An NSTextStorage object stores text attributes in dictionaries (see the NSDictionary class specification for more information). Each range of characters that share the same attributes conceptually share a dictionary. Within the dictionary, attributes are identified by a key which has an associated value. In the preceding implementation of **emphasizeText:**, the attribute we add to the selected text is identified by the globally scoped key NSFontAttributeName whose value is set to the NSFont object representing the Helvetica-Oblique type face.

Perhaps setting the font to an oblique angle doesn't provide enough emphasis, so you decide to additionally have the text drawn in blue on a red background. You can accomplish this by sending two more **addAttributeValue:range:** messages, in which case the **beginEditing** and **endEditing** messages are required, and not merely good coding practice. However, since you plan to use this set of attributes repeatedly, a better idea is to create a dictionary containing this set. This dictionary defines a style that you can use repeatedly:

```
NSDictionary *emphasisAttributes = [NSDictionary
    dictionaryWithObjectsAndKeys:
        [NSColor blueColor],NSForegroundColorAttributeName,
        [NSColor redColor], NSBackgroundColorAttributeName,
        [NSFont fontWithName:@"Helvetica-Oblique" size:12.0],
        NSFontAttributeName, nil];


- (void)emphasizeText:(id)sender
```

```
{
    NSTextStorage *theTextStore = [theTextView textStorage];
    NSRange changeCharRange = [theTextView
        rangeForUserCharacterAttributeChange];


    if (changeCharRange.location == NSNotFound) return;


    if ([theTextView shouldChangeTextInRange:changeCharRange
                    replacementString:nil]) {
        [theTextStore beginEditing];


        [theTextStore addAttributes:emphasisAttributes
            range:changeCharRange];


        [theTextStore endEditing];
        [theTextView didChangeText];
    }
    return;
```

Note the use of the **addAttributes:range:** method. This method is similar to the **addAttribute:range:** method, but applies a dictionary of attributes rather than a single attribute. With either method, an added attribute replaces an existing one. For example, if the foreground color is set to green and you then invoke the **emphasizeText:** method above, the new value of the foreground color is blue. Of course, this is the correct behavior and is a result of storing attributes in a dictionary, where a given key can have only one value.

Characters and Glyphs

Characters are conceptual entities that correspond to units of written language. Examples of characters include the letters of the Roman alphabet, the Kanji ideographs used in Japanese, and symbols that indicate mathematical operations. Characters are represented as numbers in a computer's memory or on disk, and a *character encoding* defines the mapping between a numerical value and a specific character. For example, the ASCII and Unicode character encodings both assign the value 97 (decimal) to the character 'a'. The OPENSTEP text-handling system uses the Unicode character encoding internally, although it can read and write other encodings on disk.

You can think of a *glyph* as the rendered image of a character. The words of this sentence are made visible through glyphs. A collection of glyphs that share certain graphic qualities is called a *font*.

The difference between a character and a glyph isn't immediately apparent in English since there's typically a one-to-one mapping between the two. But, in some Indic languages, for example, a single character can map to more than one glyph. And, in many languages, two or more characters may be needed to specify a single glyph. To take a simple example, the glyph 'ö' can be the result of two characters, one representing the base character 'o' and the other representing the diacritical mark '¨'. A user of a word processor can strike the arrow key one time to move the insertion point from one side of the 'ö' glyph to the other; however, the current position in the character stream must be incremented by two to account for the two characters that make up the single glyph.

Thus, the text system must manage two related but different streams of data: the stream of characters (and their attributes) and the stream of glyphs that are derived from these characters. The NSTextStorage object stores the attributed characters, and the NSLayoutManager stores the derived glyphs. Finding the correspondence between these two streams is another responsibility of the NSLayoutManager.

For a given glyph the NSLayoutManager can find the corresponding character or characters in the character stream. Similarly, for a given character, the NSLayoutManager can locate the associated glyph or glyphs. For example, when a user selects a range of text, the NSLayoutManager must determine which range of characters corresponds to the selection.

When characters are deleted, some glyphs may have to be redrawn. For example, if the user deletes the characters "ee" from the word "feel", the 'f' and 'l' can be represented by the 'fl' ligature rather than the two glyphs 'f' and 'l'. The NSLayoutManager has new glyphs generated as needed. Once the glyphs are regenerated, the text must be laid out and displayed. Again, the NSLayoutManager is instrumental in this step. Working with the NSTextContainer and other objects of the text system, the NSLayoutManager determines where each glyph appears in the NSTextView. Finally, the NSTextView renders the text.

Since an NSLayoutManager is central to the operation of the text-handling system, it also serves as the repository of information shared by various components of the system.

These are just some of the functions of an NSLayoutManager; others are discussed in later sections.

Class Clusters

The Foundation Framework'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 framework without reducing its functional richness.

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 Framework, which includes these clusters:

Class Cluster	**Public Superclasses**
NSData	NSData NSMutableData
NSArray	NSArray NSMutableArray
NSDictionary	NSDictionary NSMutableDictionary
NSString	NSString NSMutableString

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 only appropriate for cluster objects that allow their contents to be modified.

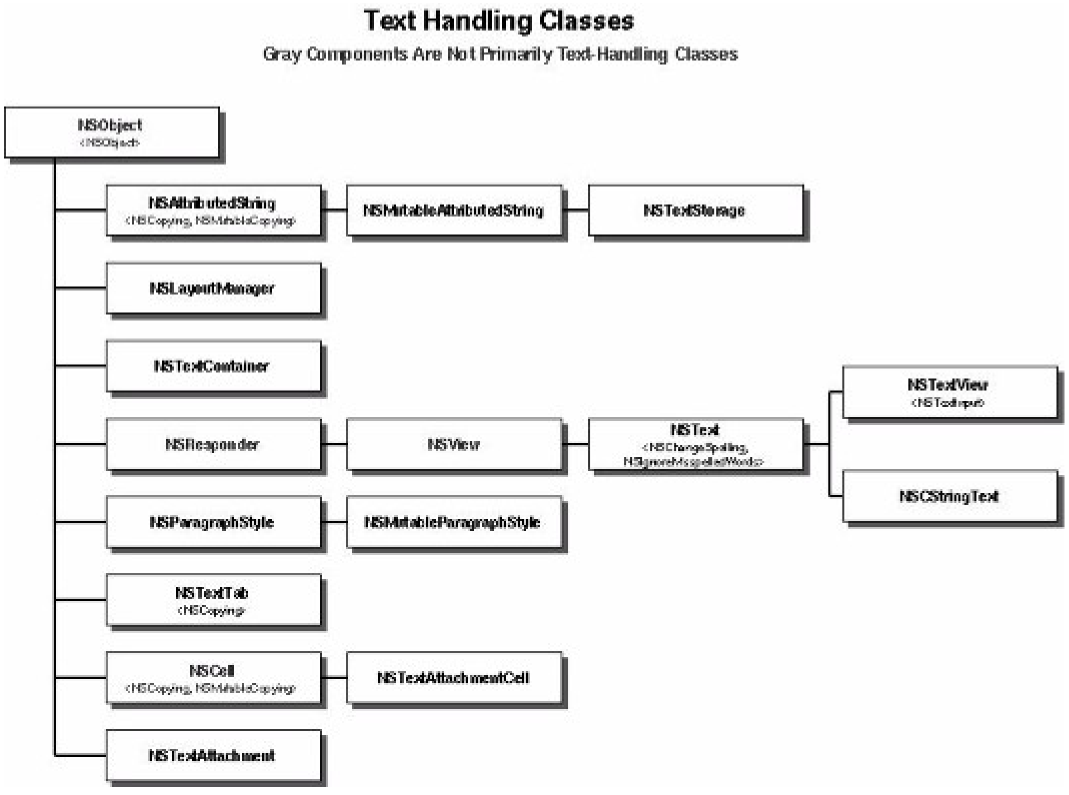
This factoring of the cluster’s interface helps make an object-oriented framework’s programmatic interface more expressive. For example, imagine a Book object that declares this method:

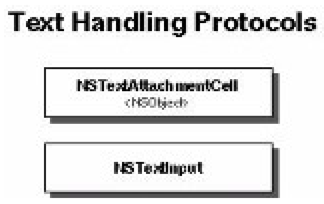
```
- (NSString *)title;
```

The book object could return its own instance variable or create a new string object and return that—it 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.

Class Hierarchy of the Text-Handling System

You’ve seen the four principal classes in the text-handling system, but there are a number of auxiliary classes and protocols that make up the system. The diagrams below give you a picture of the complete system. Following the diagrams is a synopsis of the elements that haven’t been introduced so far.



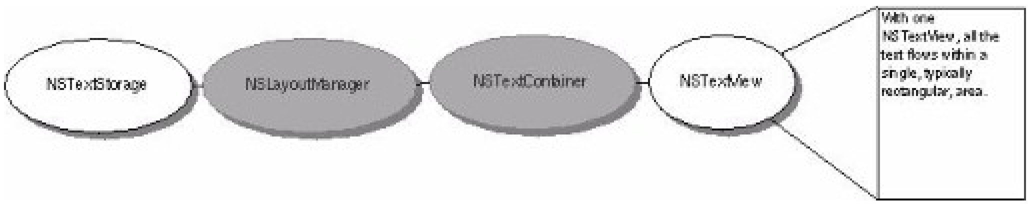


- NSFileWrapper, NSTextAttachment, and NSTextAttachmentCell
- NSTextInput protocol, NSInputManager, and NSInputServer
- NSParagraphStyle, NSMutableParagraphStyle, and NSTextTab

Common Configurations

The following diagrams give you an idea of how you can configure objects of these four classes —NSTextStorage, NSLayoutManager, NSTextContainer, and NSTextView—to accomplish different text-handling goals.

To display a single flow of text, the objects are arranged like this:

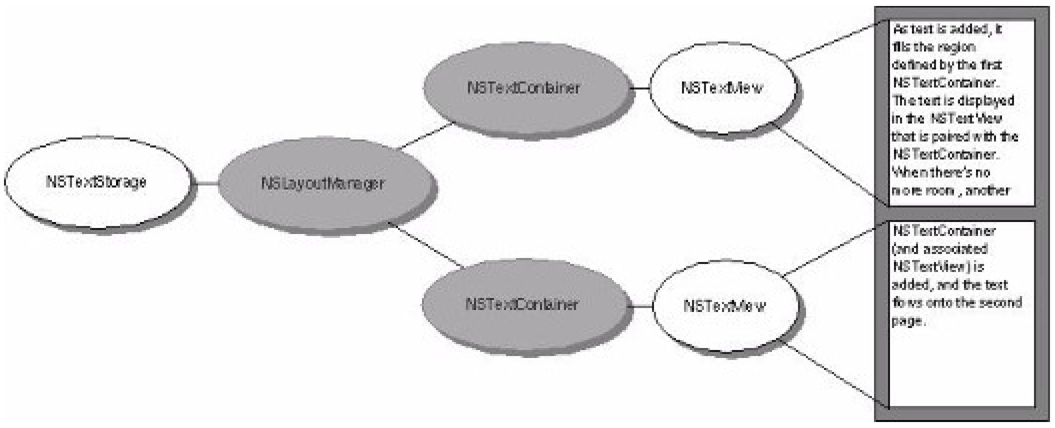

NSTextStorage
NSLayoutManager
NSTextContainer
NSTextView
With one NSTextView, all the text flows within a single, typically rectangular, area.

The NSTextView provides the view that displays the glyphs, and the NSTextContainer object defines an area within that view where the glyphs are laid out. Typically in this configuration, the NSTextContainer's vertical dimension is declared to be some extremely large value so that the container can accommodate any amount of text, while the NSTextView is set to size itself around the text using the **setVerticallyResizable:** method defined by NSText, and given a maximum height equal to the NSTextContainer's height. Then, with the NSTextView embedded in an NSScrollView, the user can scroll to see any portion of this text.

If the NSTextContainer's area is inset from the NSTextView's bounds, a margin appears around the text. The NSLayoutManager object, and other objects not pictured here, work together to generate glyphs from the NSTextStorage's data and lay them out within the area defined by the NSTextContainer.

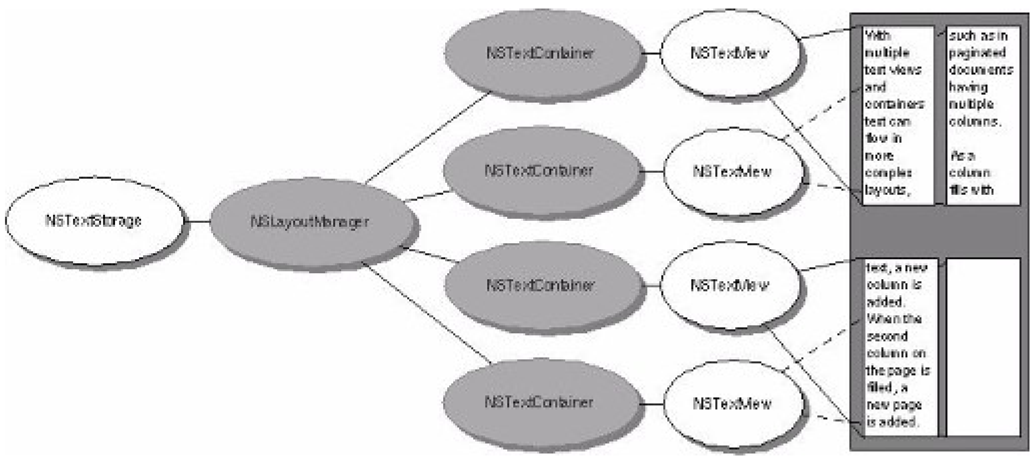
This configuration is limited by having only one NSTextContainer-NSTextView pair. In such an arrangement, the text flows uninterrupted within the area defined by the NSTextContainer. Page breaks, multi-column layout, and more complex layouts can't be accommodated by this arrangement.

By using multiple NSTextContainer-NSTextView pairs, more complex layout arrangements are possible. For example, to support page breaks, an application can configure the text-handling objects like this:


NSTextContainer
NSTextView
NSTextStorage
NSLayoutManager
NSTextContainer
NSTextView
As text is added, it fills the region defined by the first NSTextContainer. The text is displayed in the NSTextView that is paired with the NSTextContainer. When there's no more room, another
NSTextContainer (and associated NSTextView) is added, and the text flows onto the second page.

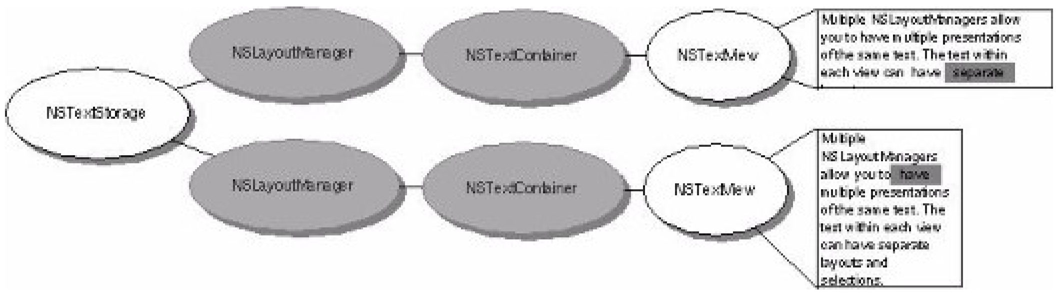
Each NSTextContainer-NSTextView pair corresponds to a page of the document. The gray rectangle in the diagram above represents a custom view object that your application provides as a background for the NSTextViews. This custom view can be embedded in an NSScrollView to allow the user to scroll through the document's pages.

A multi-column document uses a similar configuration:


NSTextStorage
NSLayoutManager
NSTextContainer
NSTextContainer
NSTextContainer
NSTextContainer
NSTextView
NSTextView
NSTextView
NSTextView
With multiple text views and containers text can flow in more complex layouts,
such as in paginated documents having multiple columns.
As a column fills with
text, a new column is added. When the second column on the page is filled, a new page is added.

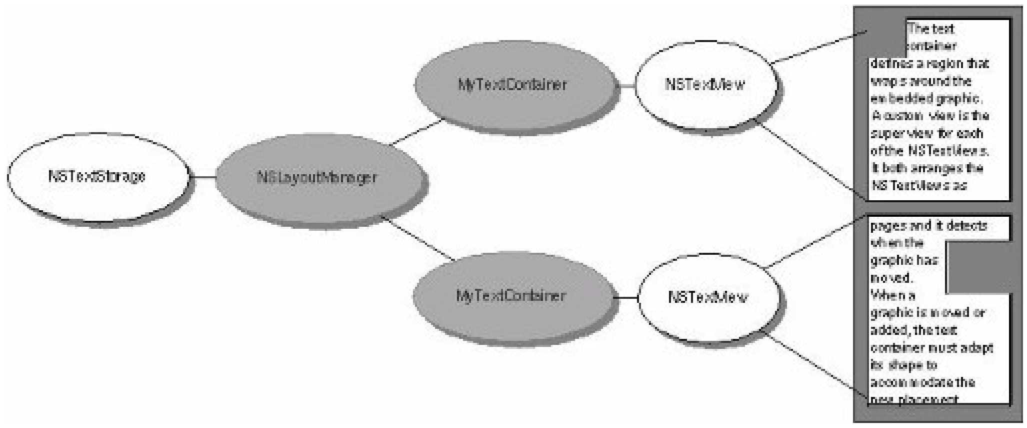
Instead of having one NSTextView-NSTextContainer pair correspond to a single page, there are now two pairs—one for each column on the page. Each NSTextContainer-NSTextView controls a portion of the document. As the text is displayed, glyphs are first laid out in the top-left view. When there is no more room in that view, the NSLayoutManager informs its delegate that it has finished filling the container. The delegate can check whether there's more text that needs to be laid out and add another NSTextContainer and NSTextView. The NSLayoutManager proceeds to lay out text in the next container, notifies the delegate when finished, and so on. Again, a custom view (depicted as a gray rectangle) provides a canvas for these text columns.

Not only can you have multiple NSTextContainer-NSTextView pairs, you can also have multiple NSLayoutManagers accessing the same NSTextStorage. The simplest arrangement looks like this:


NSTextStorage
NSLayoutManager
NSLayoutManager
NSTextContainer
NSTextContainer
NSTextView
NSTextView
Multiple NSLayoutManagers allow you to have multiple presentations of the same text. The text within each view can have separate
Multiple NSLayoutManagers allow you to have multiple presentations of the same text. The text within each view can have separate layouts and selections.

The effect of this arrangement is to give multiple views on the same text. If the user alters the text in the top view, the change is immediately reflected in the bottom view (assuming the location of the change is within the bottom view's bounds).

Finally, complex page layout requirements, such as permitting text to wrap around embedded graphics, can be achieved by a configuration that uses a custom subclass of NSTextContainer. This subclass defines a region that adapts its shape to accommodate the graphic image:


NSTextStorage
NSLayoutManager
MyTextContainer
MyTextContainer
NSTextView
NSTextView
The text container defines a region that wraps around the embedded graphic. A custom view is the super view for each of the NSTextViews. It both arranges the NSTextViews as
pages and it detects when the graphic has moved. When a graphic is moved or added, the text container must adapt its shape to accommodate the new placement

Contents

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 invoke—you don't, and can't, choose the class of the instance.

In the Foundation Framework, you generally create an object by invoking a + ***className...*** method or the **alloc...** and **init...** methods. Taking the Foundation Framework'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 automatically—See "Object Ownership and Automatic Disposal" 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 returned—*aChar*, *anInt*, *aFloat*, and *aDouble*—may 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. Although it is not precisely correct, it's convenient to 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 method declared by NSNumber.

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 framework 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 Framework 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 a 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 Framework 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 Framework object in an object of your own design. The sections below give more detail on these two approaches.

Creating an NSTextView Object

All applications use the text-handling system, if only to display the titles of buttons and other labels. Most applications have far greater need of the system than that. This section describes the most direct ways of assembling the network of objects that make up that system.

Creating an NSTextView Programmatically

At times, you may need to assemble the text-handling system programmatically. You can do this in either of two ways: by creating an NSTextView object and letting it create its network of supporting objects or by building the network of objects yourself. In most cases, you'll find it sufficient to create an NSTextView object and let it create the underlying network of text-handling objects, as discussed in this section. If your application has complex text-layout requirements, you'll have to create the network yourself; see "Assembling the Text System by Hand" for information.

You create an NSTextView object in the usual way: by sending the **alloc** and **init...** messages. Given an NSWindow object represented by **aWindow**, you can create an NSTextView object in this way:

```
/* determine the size for the NSTextView */
NSRect cFrame =[[aWindow contentView] frame];


/* create the NSTextView and add it to the window */
NSTextView *theTextView = [[NSTextView alloc] initWithFrame:cFrame];

[aWindow setContentView:theTextView];
[aWindow makeKeyAndOrderFront:nil];
[aWindow makeFirstResponder:theTextView];
```

This code determines the size for the NSTextView's frame rectangle by asking **aWindow** for the size of its content view. The NSTextView is then created and made **aWindow**'s content view using **setContentView:**. Finally, the **makeKeyAndOrderFront:** and **makeFirstResponder:** messages display the window and cause **theTextView** to prepare to accept keyboard input.

NSTextView's **initWithFrame:** method not only initializes the receiving NSTextView object, it causes the object to create and interconnect the other components of the text-handling system. This is a convenience that frees you from having to create and interconnect them yourself. Since the NSTextView created these supporting objects, it's responsible for releasing them when they are no longer needed. When you're done with the NSTextView, release it and it takes care of releasing the other objects of the text-handling system. Note that this ownership policy is only in effect if you let NSTextView create the components of the text-handling system. See "Assembling the Text System by Hand" for more information on object ownership when you create the components yourself.

Enabling Services Menu Items Based on the Selection

While your application is running, various types of data can be selected and available for transfer on the pasteboard. If a service doesn't apply to the type of the selected data, its menu item needs to be disabled. To check whether a service applies, the application object sends **validRequestorForSendType:returnType:** messages to objects in the responder chain to see whether they have data of the type used by that service. While the Services menu is visible, this method is invoked frequently—typically many times per event—to ensure that the menu items for all service providers are properly enabled: It's sent for each service and possibly for many objects in the responder chain. Because this method is invoked so frequently, it must be fast so that event handling doesn't fall behind the user's actions.

The following example shows how this method can be implemented for an object that handles unformatted text:

```
- (id)validRequestorForSendType:(NSString *)sendType
    returnType:(NSString *)returnType;
{

    if ( (!sendType || [sendType isEqual:NSStringPboardType]) &&
         (!returnType || [returnType isEqual:NSStringPboardType]) ) {

        if ( ([self selection] || !sendType) &&
             ([self isEditable] || !returnType) ) {
            return self;
        }
    }


    return [super validRequestorForSendType:sendType
        returnType:returnType];

}
```

This implementation checks both the types indicated and the state of the object. The object is a valid requestor if the send and return types are unformatted text or simply aren't specified, and if the object has a selection and is editable (when send and return types are given). If this object can't handle the service request in its current state, it invokes its superclass' implementation.

validRequestorForSendType:returnType: is sent along an abridged responder chain, comprising only the responder chain for the key window and the application object. The main window is excluded.

Entries in a Service Specification

This template shows all possible fields in a standard service specification:

```
NSServices = (
    {   NSMessage = messageName;
        NSPortName = programName;
        NSSendTypes = ( type1 [, type2] ... );
        NSReturnTypes = ( type1 [, type2] ... );
        NSMenuItem = { default = item; [ language = item; ] };
        NSKeyEquivalent = { default = character; [ language = character; ] };
        NSUserData = string;
        NSTimeout = milliseconds;
        NSHost = hostName;
        NSExecutable = pathname;
    }
    [, { another service entry } ] ...
);
```

Filter, print filter, and spell checker services differ slightly. Their service specifications are described in
"Variations on Standard Services."

NSMessage indicates the name of the Objective-C method to invoke. Its value is the first part of the method name, which follows the form *messageName***:userData:error:**. This is a required entry.

NSPortName is the name of the port the application should use to listen for service requests. Its value depends on how you registered the service provider. If you used the NSApplication method **setServicesProvider:**, **NSPortName** is the application name. If you used the **NSRegisterServicesProvider()** function (which should only be used for filter services), **NSPortName** is the value passed to that function for its *name* argument. See "Filter Services" for more information on **NSRegisterServicesProvider()**. This is a required entry.

NSSendTypes and **NSReturnTypes** are arrays of names for data types, such as **NSStringPboardType**. Send types are the types sent from the service requestor; return types are the types returned to the service requestor. See the NSPasteboard class specification for a list of standard data types. A service provider must specify one or both of these entries.

NSMenuItem and **NSKeyEquivalent** indicate the text of the Services menu item and its key equivalent (if any). Both of these entries take the form of dictionaries, with language names as keys and the text as values. In addition to actual language names, you can define a value for the key **default**, which is used when no languages in the user's preferences match the languages named in the service specification. The text of a menu item can indicate a single submenu with a slash; for example, "Mail/Send Selection" appears in the Services menu as a submenu named "Mail" with an item named "Send Selection". **NSMenuItem** is required, but **NSKeyEquivalent** is optional.

NSUserData is a string containing a value of your choice. You can use this string to control the behavior of your service method; this entry is useful for applications that provide open-ended services (see "Add-on Services"). **NSUserData** is an optional entry.

NSTimeout is a string indicating the number of milliseconds the Services facility should wait for a response from the service provider when a response is required. If this time is exceeded, the services facility opens an attention panel informing the user that an error has occurred. This is an optional entry. If you don’t specify this entry, the timeout value is 3000 milliseconds (30 seconds).

NSHost is a string containing the name of a host on the network. The executable is launched on this host instead of on the host of the application requesting the service. This is an optional entry.

NSExecutable is the path of the application that performs this service. This can either be a full or relative path. If it is a relative path, the application must be located in the same bundle as this service declaration. This entry is most useful for filter services. This entry is optional.

Filter Services

The NSPasteboard class automatically uses a filter service when you invoke a method for filtering data, such as:

+ (NSArray *)**typesFilterableTo:**(NSString *)*type*
+ (NSPasteboard *)**pasteboardByFilteringFile:**(NSString *)*filename*
+ (NSPasteboard *)**pasteboardByFilteringData:**(NSData *)data
  ofType:(NSString *)*type*
+ (NSPasteboard *)**pasteboardByFilteringTypesInPasteboard:**
  (NSPasteboard *)*pboard*

Because filter services commonly translate data from unknown file formats into known formats, you need a way of dynamically specifying pasteboard types. The filter services and pasteboard facilities define types based on file extensions with these functions:

NSString ***NSCreateFilenamePboardType(**NSString **fileExtension***)**
NSString ***NSCreateFileContentsPboardType(**NSString **fileExtension***)**
NSString ***NSGetFileType(**NSString **pboardType***)**
NSArray ***NSGetFileTypes(**NSArray **pboardTypes***)**

The *fileExtension* argument is a file extension, minus the period (for example, "eps" or "tiff"). You create pasteboard type strings with the first two functions, and get file types (extensions) from pasteboard type strings with the second two functions. In a service specification (in the **CustomInfo.plist** file), you can indicate a file type based on the extension as **NSTypedFilenamesPboardType:***fileExtension* and a file contents type as **NSTypedFileContentsPboardType:***fileExtension*; for example:

```
NSSendTypes = (NSTypedFilenamesPboardType:tiff);

NSSendTypes = (NSTypedFileContentsPboardType:tiff);
```

You implement a filter service exactly like a standard service, with a *filterName***:userData:error:** method that accepts a pasteboard containing a file path, converts the contents of the file to the requested type or types, and returns the converted data on the pasteboard. There are two major differences between filter services and standard services. The first major difference is in the way you register the service provider. With filter services, you typically don't have an NSApplication object to register the service provider with. Instead, you use the function **NSRegisterServicesProvider()**. This function's declaration is:

(void)**NSRegisterServicesProvider(**id *provider*, NSString **name***)**

provider is the object that provides the services, and *name* is the same value you specify for the **NSPortName** entry in the services specification. After making this function call, the filter service must enter the run loop in order to respond to service requests as shown:

```
while(1) {

  NS_DURING

     [[NSRunLoop currentRunLoop] run];

  NS_HANDLER
```

```
        NSLog(@"Received exception: %@", localException);

    NS_ENDHANDLER

}
```

The second major difference is in the service specification: Instead of an **NSMessage** entry you define an **NSFilter** entry with *filterName* as the value; you must define both send and return types; and the **NSMenuItem** and **NSKeyEquivalent** entries are ignored.

A filter service can use data-transfer mechanisms other than the pasteboard, indicated by an optional entry in the filter service specification. The key is **NSInputMechanism**, and it can have a value of **NSUnixStdio**, **NSMapFile**, or **NSIdentity**. If you specify an input mechanism, the value for the **NSFilter** entry is ignored (though it's still required).

NSUnixStdio allows you to turn nearly any UNIX command-line program into a filter service. Instead of sending an Objective-C message to an object in your filter service program, the services facility simply runs the executable specified in the service specification with the contents of the pasteboard as the argument (which must be of **NSFilenamesPboardType** or **NSTypedFilenamesPboardType**). If there is more than one filename on the pasteboard, only the first is used. The output of the filter program (on **stdout**) is captured by the services facility and put on a pasteboard for use by the requestor of the filter. Note that the UNIX program must be relaunched every time the service is invoked; if you're creating a filter service from scratch it's more efficient to package it as an application that can remain running. Here's a sample service specification for a UNIX program that converts GIF images to TIFF:

```
{
    NSServices = (
      { NSFilter = "";
        NSPortName = gif2tiff;
        NSInputMechanism = NSUnixStdio;
        NSSendTypes = (NSTypedFilenamesPboardType:gif);
        NSReturnTypes = (NSTIFFPboardType);
      }
    );
}
```

NSMapFile defines an "empty" service for data in files, used when you invoke NSPasteboard's **pasteboardByFilteringFile:** class method. Its value must be an **NSFilenamesPboardType** or an **NSTypedFilenamesPboardType**. When the filter service is invoked for a file, the services facility merely puts the contents of the file on the pasteboard. This input mechanism is useful for file types with nonstandard or special extensions whose format is nonetheless the same as a standard type. For example, if you've defined an image format based on a subset of TIFF and given it a file extension of **stif**, you can define a service that maps the **stif** file extension to **NSTIFFPboardType**:

```
{
    NSServices = (
      { NSFilter = "";
        NSInputMechanism = NSMapFile;
        NSSendTypes = (NSTypedFilenamesPboardType:stif);
        NSReturnTypes = (NSTIFFPboardType);
      }
    );
}
```

NSIdentity defines an empty service for data in memory, used when you invoke NSPasteboard's **pasteboardByFilteringData:ofType:** class method. It declares that the send type is effectively identical to the return type—though the reverse isn't necessarily true. For example, you can define a service that filters your custom image format in memory with this service specification:

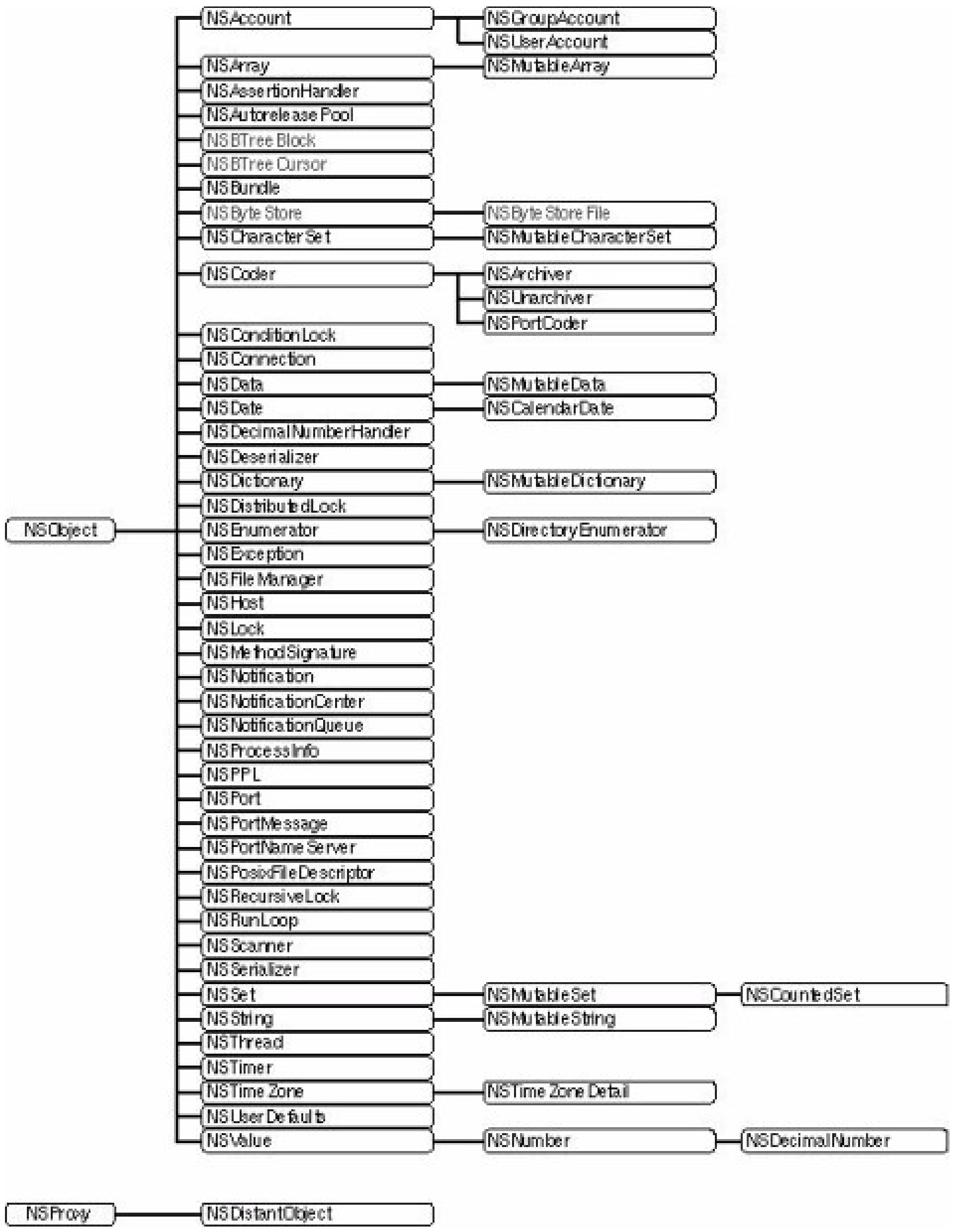
```
{
    NSServices = (
      { NSFilter = "";
        NSInputMechanism = NSIdentity;
        NSSendTypes = (MyCustomImagePboardType);
        NSReturnTypes = (NSTIFFPboardType);
      }
    );
}
```

Neither **NSMapFile** nor **NSIdentity** result in any program being executed, so their services specifications lack the **NSPortName** entry.

Foundation Framework Classes

The OpenStep class hierarchy is rooted in the Foundation Framework's NSObject class. The remainder of the Foundation Framework consists of several related groups of classes as well as a few individuals. Most of the groups form what are called *class clusters*—abstract 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 "Class Clusters" for a full treatment of this new concept.

The complete Foundation Framework class inheritance hierarchy looks like this:



Many of these classes have closely related functionality:

- **Data storage** NSData and NSString provide object-oriented storage for arrays of bytes. NSValue and NSNumber provide object-oriented storage for arrays of simple C data values. NSArray, NSDictionary, NSPPL, and NSSet provide storage for Objective-C objects of any class.

- **Text and strings**. NSCharacterSet represents various groupings of characters which are used by the NSString and NSScanner classes. The NSString classes represent text strings and provide methods for searching, combining, and comparing strings. An NSScanner object is used to scan numbers and words from an NSString object.

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

- **Application coordination and timing.** NSNotification, NSNotificationCenter, and NSNotificationQueue provide systems that an object can use to notify all interested observers of changes that occur. You can use a NSTimer object to send a message to another object at specific intervals.

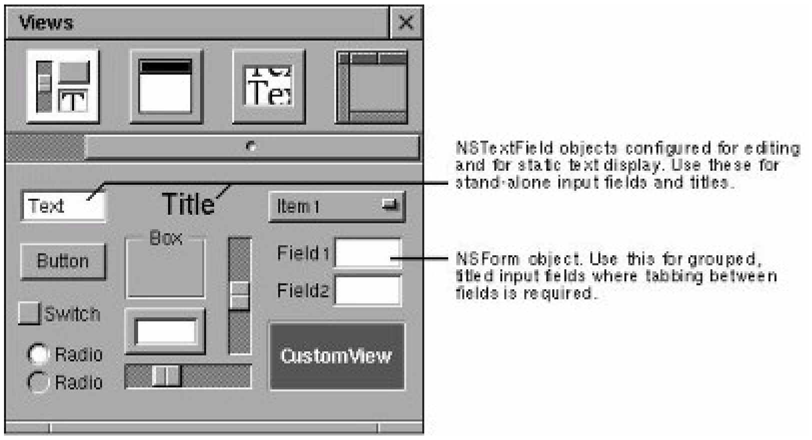
- **Object creation and disposal**. NSAutoreleasePools are used to implement the delayed-release feature of the Foundation Framework, as described in "Object Ownership and Automatic Disposal."

- **Object distribution and persistence.** The data that an object contains can be represented in an architecture-independent way using NSSerializer. NSCoder and its subclasses take this process a step further by allowing class information to be stored along with the data. The resulting representations are used for archiving and for object distribution.

- **Operating system services.** Several classes are designed to insulate you from the idiosyncracies of various operating systems. The NSAccount, NSUserAccount, NSGroupAccount, and NSHost classes provide an object-oriented representation of various account and host data. NSFileManager provides a consistent interface for file operations (creating, renaming, deleting, and so on). NSThread and NSProcessInfo let you create multi-threaded applications and query the environment in which an application runs.

Interface Builder and the Text-Handling System

The easiest way to use the text-handling system is through the objects on Interface Builder's palettes. The control objects (NSForm and NSTextField) provide objects that are preconfigured for specific uses:

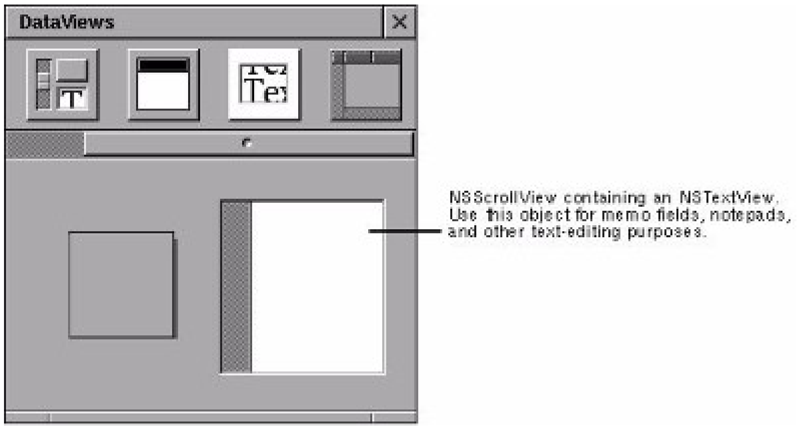


NSTextField objects configured for editing and for static text display. Use these for stand-alone input fields and titles.

NSForm object. Use this for grouped, titled input fields where tabbing between fields is required.

Using Interface Builder's Inspector panel, you can set many text-related attributes of these controls. For example, you can specify whether the text in a text field is selectable, editable, scrollable, and so on. The Inspector panel also lets you set the text alignment and background and foreground colors.

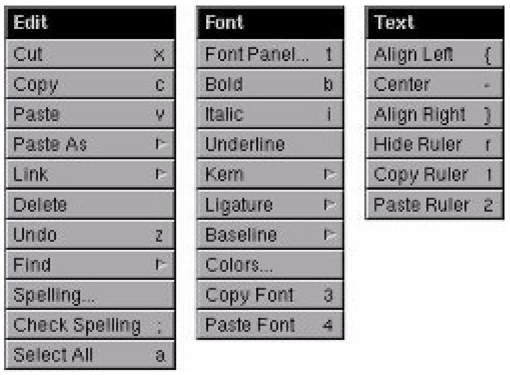
Interface Builder also provides a scrolling text view that supports the features of a basic text editor:



NSScrollView containing an NSTextView. Use this object for memo fields, notepads, and other text-editing purposes.

The NSScrollView inspector in Interface Builder lets you specify, among other things, whether the contained NSTextView allows multiple fonts and embedded graphics.

Much more of NSTextView's functionality is accessible through menu commands. Interface Builder's Palettes window offers these ready-made menus that contain text-related commands:



By default, most of the commands in these menus operate on the *first responder*, that is, the view within the key window that the user has selected for input. (See the NSResponder, NSView, and NSWindow class specifications for more information on the first responder.) In practice, the first responder is the object that's displaying the selection, say a drawing object in the case of a graphical selection or an NSTextView in the case of a textual selection. By adding these menus to your application, you can offer the user access to many powerful text-editing features.

NSTextViews cooperate with the Services facility through the Services menu, also available from Interface Builder's Menus palette. By simply adding the Services menu item to your application's main menu, the NSTextViews in your application can access services provided by other applications. For example, if the user selects a word within an NSTextView and chooses the Define in Webster service, the NSTextView passes its selected text to the Webster application for look up.

Interface Builder offers these direct ways of accessing the features of the text-handling system. You can also configure your own menu items or other controls within Interface Builder to send messages to an NSTextView. For example, you can make an NSTextView output its text for printing or faxing by sending it a **print:** or **fax:** message. One way to do this is to drag a menu item from Interface Builder's Menu palette into your application's main menu and hook it up to an NSTextView (either through the first responder or by direct connection). By specifying that the item send a **print:** message to its target, the NSTextView's contents can be printed or faxed when the application is run.

Invoking a Standard Service Programmatically

Though the user typically invokes a standard service by choosing an item in the Services menu, you can invoke it in code using this function:

BOOL **NSPerformService(**NSString **serviceItem*, NSPasteboard **pboard***)**

This function returns **YES** if the service is successfully performed, **NO** otherwise. *serviceItem* is the name of a Services menu item (in any language). It must be the full name of the service, including the submenu and slash; for example, “Mail/Selection”. *pboard* contains the data to be used for the service, and when the function returns contains the data resulting from the service. You can then do with the data what you wish.

Key Bindings

The new text system uses a generalized key binding mechanism which is completely remappable by the user. The standard bindings for can always be found in **NextLibrary/Frameworks/AppKit.framework/Resources/StandardKeyBinding.dict** or in **/NextLibrary/Frameworks/AppKit.framework/Resources/StandardKeyBinding-winnt.dict**. On both platforms these standard bindings include a large number of Emacs-compatible control key bindings, all the various arrow key bindings, bindings for making field editors and some keyboard UI work, and backstop bindings for many function keys. On Windows the standard bindings also include a number of Emacs-compatibile Alt key bindings (like Alt-f, Alt-b).

All these bindings are customizable by the user. You can create a file in **~/Library/KeyBindings/DefaultKeyBinding.dict** to augment or replace the standard bindings. Use the standard bindings files as templates. Modifier flags are specified using special characters: "^" for control, "~" for Alt, "$" for Shift, and "#" for numeric keypad. Multiple keystroke bindings are supported through nested binding dictionaries. For instance, Escape could be bound to "cancel:" or it could be bound to a whole dictionary which would then contain bindings for the next keystroke after Escape.

Here are a couple sample binding files that you might use:

1. The first one adds Alt-key bindings for some common Emacs stuff. This might be useful on Mach where the Alt-key bindings are not standard. With these bindings it would be necessary to type "Control-Q, Alt-f" in order to type a florin character instead of moving forward a word. This sample also explicitly binds Escape to "complete:". On Mach, this is the default so this override changes nothing, but on Windows, Escape is bound to "cancel:" by default, so this example changes it so Escape will mean **complete:** when a text object is key (it will still mean **cancel:** if some non-textual thing, like an NSButton, is key).

```
/* ~/Library/KeyBindings/DefaultKeyBinding.dict */

{
     /* Additional Emacs bindings */
     "~f" = "moveWordForward:";
     "~b" = "moveWordBackward:";
     "~<" = "moveToBeginningOfDocument:";
     "~>" = "moveToEndOfDocument:";
     "~v" = "pageUp:";
     "~d" = "deleteWordForward:";
     "~^h" = "deleteWordBackward:";
     "~\010" = "deleteWordBackward:";  /* Alt-backspace */
     "~\177" = "deleteWordBackward:";  /* Alt-delete */

     /* Escape should really be complete: */
     "\033" = "complete:";  /* Escape */
}
```

2. This example shows how to have multi-keystroke bindings. It binds a number of Emacs meta bindings using Escape as the meta key instead of the Alt modifier. So Escape followed by f means **moveWordForward:** here. This sample binds Esc-Esc to "complete:". Note the nested dictionaries.

```
/* ~/Library/KeyBindings/DefaultKeyBinding.dict */

{
     /* Additional Emacs bindings */
     "\033" = {
            "\033" = "complete:";  /* ESC-ESC */
            "f" = "moveWordForward:";  /* ESC-f */
            "b" = "moveWordBackward:";  /* ESC-b */
            "<" = "moveToBeginningOfDocument:";  /* ESC-< */
            ">" = "moveToEndOfDocument:";  /* ESC-> */
            "v" = "pageUp:";  /* ESC-v */
            "d" = "deleteWordForward:";  /* ESC-d */
            "^h" = "deleteWordBackward:";  /* ESC-Ctrl-H */
            "\010" = "deleteWordBackward:";  /* ESC-backspace */
            "\177" = "deleteWordBackward:";  /* ESC-delete */
        };
    }
```

With the right combination of key bindings and default settings, it should be possible to tailor the text system to your preferences.

Making a Service Available

Now you have an object with methods that allow it to perform a service for another application. There are two things remaining to do: register the object at run time so the services facility knows which object to have perform the service, and advertise the service to the services facility. You create and register your object in the **applicationDidFinishLaunching:** application delegate method (or equivalent) with NSApplication's **setServicesProvider:** method. If your object is called **encryptor** you create and register it with this code fragment:

```
EncryptoClass *encryptor;

encryptor = [[EncryptoClass alloc] init];
[NSApp setServicesProvider:encryptor];
```

You can register only one service provider per application. If you have more than one service to provide, a single object must be able to provide all of the services.

In order for the system to know that your application provides a service, you must advertise that fact. You do this by adding an entry to your application project's **CustomInfo.plist** file, which is incorporated into the application's **Info.plist** file when you build your project. The entry you add is called the *service specification*. In our example, the service specification looks like this:

```
{
    NSServices = (
      { NSPortName = NewsReader;
        NSMessage = simpleEncrypt;
        NSSendTypes = (NSStringPboardType);
        NSReturnTypes = (NSStringPboardType);
        NSMenuItem = {
            default = "Encrypt Text";
            English = "Encrypt Text";
            French = "Encoder le texte";
            German = "Text verschlüsseln";
        };
        NSKeyEquivalent = {
            default = E;
            German = S;
        };

      }
    );
}
```

The meaning of each of the subfields is explained further in "Entries in a Service Specification."

Note: If you've just built an application with a service and you want to test the service, log out and log back in again. The application must be in one of the standard directories: **~/Apps**, **/NextApps**, or **/LocalApps**.

Marking Objects for Disposal

The **autorelease** method, defined by NSObject, marks the receiver for later release. By autoreleasing an object—that is, by sending it an **autorelease** message—you 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 "Validity of Shared Objects"). 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 below).

Object Ownership and Automatic Disposal

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 ownership—and responsibility for disposal—blur. 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 Framework 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 Framework provides just such a mechanism.

Object Ownership: Summary

Now that the concepts behind the Foundation Framework'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.

Print Filter Services

A print filter service is invoked when the user saves a file as a PostScript file through the Print panel. When the user clicks the Save... button on the Print panel a Save panel opens with a pop-up list near the bottom. This pop-up list contains special types of PostScript that the user can choose from. A print filter service adds an entry to this list.

You implement a print filter service as a UNIX command line program that reads PostScript on the standard input stream and writes it to a file specified on the command line by a **-o** option; for example:

```
ps2superps -o outputfile.ps
```

Instead of an **NSMessage** entry, the service specification for a print filter service contains a **NSPrintFilter** entry, whose value is the extension used for the output file. If it's empty "ps" is used by default. The **NSPortName** entry is the name of the UNIX program—**ps2superps** in the example. **NSMenuItem** gives the string that appears in the pop-up list. The following entries are ignored in a print filter service specification:

NSKeyEquivalent
NSSendTypes
NSReturnTypes
NSUserData

A print filter service specification adds one entry: **NSDeviceDependent**. Its value is "YES" or "NO" (the default). If you specify "YES" for this entry the PostScript code sent through your print filter is specific to the type of printer chosen in the Print panel.

Here's a sample print filter service specification:

```
{
    NSServices = (
      { NSPrintFilter = "superps";
        NSPortName = ps2superps;
        NSMenuItem = {
            default = "Super PostScript for Chosen Printer";
            English = "Super PostScript for Chosen Printer";
            French =
                "Super PostScript pour l'imprimante sélectionnée";
            German = "SuperPostScript für ausgewählten Drucker";
        };
        NSDeviceDependent = "YES";
      }
    );
}
```

Providing a Standard Service

Suppose you're working on a program to read USENET news, and have an object with a method to encrypt and decrypt articles, such as the one below. News articles containing offensive material are often encrypted with this algorithm, called "rot13," in which letters are shifted halfway through the alphabet.

```
- (NSString *)rotateLettersInString:(NSString *)aString
{

    NSString *newString;
    unsigned length;
    unichar *buf;
    unsigned i;


    length = [aString length];
    buf = malloc( (length + 1) * sizeof(unichar) );
    [aString getCharacters:buf];
    buf[length] = (unichar)0;  // not really needed....

    for (i = 0; i < length; i++) {
        if (buf[i] >= (unichar)'a' && buf[i] <= (unichar) 'z') {
            buf[i] += 13;
            if (buf[i] > 'z') buf[i] -= 26;
        }

         else if (buf[i] >= (unichar)'A' &&

          buf[i] <= (unichar) 'Z') {
            buf[i] += 13;
            if (buf[i] > 'Z') buf[i] -= 26;
        }
    }

    newString = [NSString stringWithCharacters:buf length:length];
    free(buf);

    return newString;
}
```

Since this feature is generally useful as a simple encryption scheme, it can be exported to other applications. To offer this functionality as a service, write a method such as this:

```
- (void)simpleEncrypt:(NSPasteboard *)pboard
    userData:(NSString *)data
    error:(NSString **)error
{

    NSString *pboardString;
    NSString *newString;
    NSArray *types;


    types = [pboard types];

    if (![types containsObject:NSStringPboardType]) {
        *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                   @"pboard couldn't give string.");
        return;
    }

    pboardString = [pboard stringForType:NSStringPboardType];
    if (!pboardString) {
        *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                     @"pboard couldn't give string.");
        return;
    }

    newString = [self rotateLettersInString:pboardString];

    if (!newString) {
        *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                     @"self couldn't rotate letters.");
        return;
    }

    types = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:types owner:nil];
    [pboard setString:newString forType:NSStringPboardType];

    return;
}
```

A method for providing a standard service is of the form *serviceName***:userData:error:** and takes arguments as shown in the example. The method itself takes data from the pasteboard as needed, operates on it, and writes any results back to the pasteboard. In case of an error, the method simply sets the pointer given by the *error* argument to a non-**nil** NSString and returns. The *userData* argument isn't used here; see "Entries in a Service Specification" and "Add-on Services" for some suggestions on how to use it.

Putting an NSTextView Object in an NSScrollView

A scrolling text view is commonly required in applications, and Interface Builder provides an NSTextView configured just for this purpose. However, at times you may need to create a scrolling text view programmatically, so it’s important to understand how to proceed.

The process consists of three steps: setting up the NSScrollView, setting up the NSTextView, and assembling the pieces.

Assuming an object has the variable **theWindow** that represents the window where the scrolling view is displayed, you can set up the NSScrollView like this:

```
NSScrollView *scrollview = [[NSScrollView alloc]
    initWithFrame:[[theWindow contentView] frame]];
NSSize contentSize = [scrollview contentSize];


[scrollview setBorderType:NSNoBorder];
[scrollview setHasVerticalScroller:YES];
[scrollview setHasHorizontalScroller:NO];
[scrollview setAutoresizingMask:NSViewWidthSizable |
    NSViewHeightSizable];
```

Note that we create an NSScrollView that completely covers the content area of the window it’s displayed in. We also specify a vertical scroll bar but no horizontal scroll bar, since this scrolling text view wraps text within the horizontal extent of the NSTextView, but lets text flow beyond the vertical extent of the NSTextView.

Finally, we set how the NSScrollView reacts when the window it’s displayed in changes size. By turning on the NSViewWidthSizable and NSViewHeightSizable bits of its resizing mask, we ensure that the NSScrollView grows and shrinks to match the window’s dimensions.

The next step is to create and configure an NSTextView to fit in the NSScrollView:

```
theTextView = [[NSTextView alloc] initWithFrame:NSMakeRect(0, 0,
    contentSize.width, contentSize.height)];
[theTextView setMinSize:NSMakeSize(0.0, contentSize.height)];
[theTextView setMaxSize:NSMakeSize(1e7, 1e7)];
[theTextView setVerticallyResizable:YES];
[theTextView setHorizontallyResizable:NO];
[theTextView setAutoresizingMask:NSViewWidthSizable];

[[theTextView textContainer] setContainerSize:contentSize];
[[theTextView textContainer] setWidthTracksTextView:YES];
```

We specify that the NSTextView’s width and height initially match those of the content area of the NSScrollView. The **setMinSize:** message tells the NSTextView that it can get arbitrarily small in width, but no smaller than its initial height. The **setMaxSize:** message allows the receiver to grow arbitrarily big in either dimension. These limits are used by the NSLayoutManager when it resizes the NSTextView to fit the text laid out.

The next three messages determine how the NSTextView’s dimensions change in response to additions or deletions of text and to changes in the scroll view’s size. The NSTextView is set to grow vertically as text is added but not horizontally. Its’s resizing mask is set to allow it to change width in response to changes in its superview’s width. Since, except for the minimum and maximum values, the NSTextView's height is determined by the amount of text it has in it, we don't let its height change with that of its superview.

The last message in this step is to the NSTextContainer, not the NSTextView. It tells the NSTextContainer to resize its width according to the width of the NSTextView. Recall that the text-handling system lays out text according to the dimensions stored in NSTextContainer objects. An NSTextView provides a place for the text to be displayed, but its dimensions and those of its NSTextContainer can be quite different. The **setWidthTracksTextView:**YES message ensures that as the NSTextView is resized, the dimensions stored in its NSTextContainer are likewise resized, causing the text to be laid out within the new boundaries.

The last step is to assemble and display the pieces:

```
    [scrollview setDocumentView:theTextView];
    [theWindow setContentView:scrollview];
    [theWindow makeKeyAndOrderFront:nil];
    [theWindow makeFirstResponder:theTextView];
```

Reading Text from a File

To read text from a file, you have to first determine format of the text. To illustrate how this is done, consider an object of the custom class Controller. A Controller object is responsible for opening and closing files. It stores an NSTextView and declares a variable that records the format of the text that it reads in. Here's the interface declaration:

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

typedef enum _dataFormat {
    Unknown = 0,
    PlainText = 1,
    RichText = 2,
    RTFD = 3,
} DataFormat;


@interface Controller : NSObject
{
    DataFormat theFormat;
    NSTextView *theTextView;
}


- (void)openFile:(id)sender;
- (void)saveFile:(id)sender;
@end
```

Now, the Controller object's **openFile:** method can be implemented like this:

```
- (void)openFile:(id)sender
{
    NSOpenPanel *panel = [NSOpenPanel openPanel];


    if ([panel runModal] == NSOKButton) {
        NSString *fileName = [panel filename];
        if ([[fileName pathExtension] isEqualToString:@"rtfd"]) {
            [theTextView readRTFDFromFile:fileName];
            theFormat = RTFD;

        } else if([[fileName pathExtension] isEqualToString:@"rtf"]) {
            NSData *rtfData = [NSData dataWithContentsOfFile:fileName];
            [theTextView replaceRange:NSMakeRange(0, [[theTextView string]
                length]) withRTF:rtfData];
            theFormat = RichText;
        } else {

            NSString *fileContents = [NSString
                stringWithContentsOfFile:fileName];
            [theTextView setString:fileContents range:NSMakeRange(0,
                [[theTextView string] length])];
            theFormat = PlainText;
        }
    }
    return;
}
```

The **openFile:** method checks the file name returned by the Open panel for the extensions "rtfd" or "rtf" and uses the appropriate means of loading data for each type. Files having any other extension are loaded as plain text. Note that the Controller object records the format of the loaded data in its **theFormat** variable. This information is used to determine how the file should be saved, as discussed in the next section.

Registering User-Interface Objects for Standard Services

The Services menu doesn’t contain *every* standard service offered by other applications. For example, in a text editor a service to invert a bitmapped image is of no use and shouldn’t be offered. Which services appear in the Services menu is determined by the data types that the objects in the application—specifically the NSResponder objects—can send and receive through the pasteboard.

An NSResponder registers these data types using NSApplication’s **registerServicesMenuSendTypes:returnTypes:** method. Application Kit objects already do this, but your custom NSResponder subclass must do this in its **initialize** class method. All types used by instances of the class must be registered, even if they’re not always available; Services menu items are enabled and disabled dynamically based on what’s available at the moment, as described in “Enabling Services Menu Items Based on the Selection”.

An object doesn’t have to register the same types for both sending and receiving. Suppose you’re writing a rich text editor that can send unformatted and rich text, but can only receive unformatted text. Here’s a portion of the initialization method for the text-editor NSView subclass:

```
+ (void)initialize
{
    static BOOL initialized = NO;

    /* Make sure code only gets executed once. */
    if (initialized == YES) return;
    initialized = YES;

    sendTypes = [NSArray arrayWithObjects:NSStringPboardType,
        NSRTFPboardType, nil];
    returnTypes = [NSArray arrayWithObjects:NSStringPboardType,
        nil];
    [NSApp registerServicesMenuSendTypes:sendTypes
        returnTypes:returnTypes];

    return;
}
```

Your NSResponder object can register any pasteboard data type, public or proprietary, common or rare. If it handles the public and common types, of course, it will have access to more services. See the NSPasteboard class specification for a list of standard pasteboard data types.

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;
}
```

Sending and Receiving Data

When the user chooses a Services menu command, the responder chain is checked with **validRequestorForSendType:returnType:** and the first object that returns a value other than **nil** is called upon to handle the service request by providing data (if any is required) with a **writeSelectionToPasteboard:types:** message. You can implement this method to provide the data immediately or to provide the data only when it's actually requested. Here's an implementation for an object that writes unformatted text immediately:

```
- (BOOL)writeSelectionToPasteboard:(NSPasteboard *)pboard
    types:(NSArray *)types
{
    NSArray *typesDeclared;


    if ([types containsObject:NSStringPboardType] == NO) {
        return NO;
    }

    typesDeclared = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:typesDeclared owner:nil];
    return [pboard setString:[self selection]
        forType:NSStringPboardType];
}
```

This method returns **YES** if it successfully writes or declares any data and **NO** if it fails. If you want to provide the data only on demand—which makes sense for large amounts—you have to declare an object as the owner for the data and then make sure that object responds to **pasteboard:provideDataForType:** (as described in the NSPasteboard class specification). In such a case, the two methods look like this:

```
- (BOOL)writeSelectionToPasteboard:(NSPasteboard *)pboard
    types:(NSArray *)types
{
    NSArray *typesDeclared;


    if ([types containsObject:NSStringPboardType] == NO) {
        return NO;
    }

    typesDeclared = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:typesDeclared owner:self];
    return YES;
}


- (void)pasteboard:(NSPasteboard *)pboard
    provideDataForType:(NSString *)type
{

    [pboard setString:[self selection] forType:NSStringPboardType];
    return;
}
```

You can even write some types in **writeSelectionToPasteboard:types:** and offer the rest on demand only via **pasteboard:provideDataForType:**. Remember that the owner of a pasteboard must exist when the data is finally requested. To be safe, you should make sure the owner is an object that will never be deallocated.

Once the service requestor writes data to the pasteboard, it waits for a response as the service provider is invoked to perform the operation; if the service doesn't return data, of course, the requesting application simply continues running and none of the following applies. The service provider reads the data from the pasteboard, works on it, and then returns the result. At this point the service requestor is sent a **readSelectionFromPasteboard:** message telling it to replace the selection with whatever data came back. Our simple text object can implement this method as follows:

```
- (BOOL)readSelectionFromPasteboard:(NSPasteboard *)pboard;
{
    NSArray *types;
    unsigned index;
    NSString *theText;


    types = [pboard types];
    index = [types indexOfObject:NSStringPboardType];
    if ([types containsObject:NSStringPBoardType] == NO) {
        return NO;
    }
    theText = [pboard stringForType:NSStringPboardType];
    [self replaceSelectionWithString:theText];

    return YES;
}
```

This method returns **YES** if it successfully reads the data from the pasteboard, **NO** otherwise.

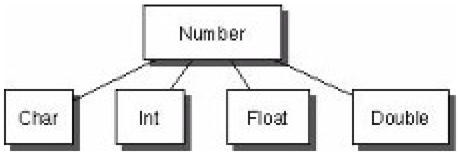
Services

The OpenStep services facility allows an application to offer its functionality to other applications, without requiring the other applications to know in advance what's offered. A service-providing application advertises an operation that it can perform on a particular type of data—for example, encrypting text, performing optical character recognition on a bitmapped image, or providing text such as a message of the day (with no input data). Any application that uses the services facility then automatically has access to that functionality through its Services menu, or through certain other mechanisms. It doesn't need to know what the operations are in advance; it merely indicates what types of data it has, and the Services menu makes available the operations that apply to those types. The services facility thus gives applications an open-ended means of extending each others' functionality.

This document describes the four available types of service: standard services, which the user chooses from the Services menu; filter services, which the developer invokes through the NSPasteboard class; print filter services, which the user chooses when saving a printout as a PostScript file; and spell checker services, which the user chooses from the standard spelling checker panel. The first section, "Standard Services," describes the general structure of all the services and the details of standard services. The second section, "Variations on Standard Services," describes ways that the other three types of service differ from standard services.

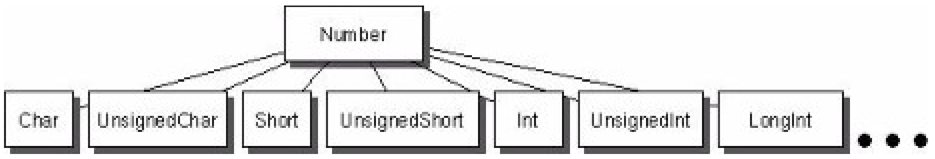
Simple Concept, Complex Interface

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 (**chars**, **ints**, **floats**, **doubles**). 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 following architecture:



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.

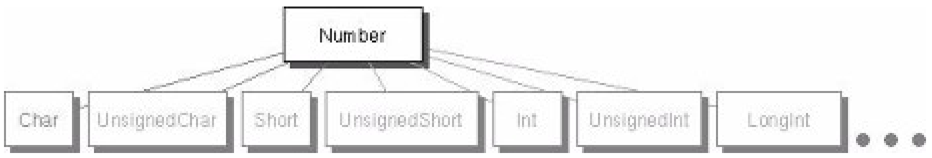
So far, this design is relatively simple. However, if the commonly used modifications of these basic C types are taken into account, the diagram looks more like this:



The simple concept—creating a class to hold number values—can 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 following hierarchy (private classes are in gray):



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.

Spell Checker Services

A Spell checker service is made available in the Application Kit's standard spell checker panel. You implement a spell checker service by creating a program that uses an NSSpellServer object. See the NSSpellServer class specification for full information on creating a spell checker service. You'll want to create the spell check service as an add-on service as described in "Add-on Services." Instead of a **NSMessage** entry, the service specification for a spell checker service contains a **NSSpellChecker** entry, whose value is the text that should be used to identify the spell checker in the spelling panel's pop-up list. A spell checker service specification should also contain a **NSLanguages** entry whose value is the language for which the spell checker applies. The spell checker won't be advertised unless one of its values for **NSLanguages** matches one of the user's preferred languages.

As an example, here's the service specification for the NeXT spell checker:

```
{
    NSServices = (
      {NSExecutable = NeXTspell;

       NSLanguages = (English);

       NSSpellChecker = NeXT;

      },
    );
}
```

Standard Services

In general terms, the standard services facility works as though the user copies data from one application and pastes it into another, modifies the data, then copies the result and pastes it back into the original application. The standard services facility does in fact use the pasteboard to transfer data, automatically copying the selection from the service requestor and pasting the altered data back—though the data transfer doesn't have to be two-way, as the examples in the introduction indicate. You should be familiar with the Application Kit's NSPasteboard class before working with the standard services facility.

This section describes how to provide a service in your application, and how to make sure your application can also request appropriate services in any situation. "Providing a Standard Service" covers everything you need to know as the implementor of a service. "Using Services" shows you what you need to make your custom classes work as requestors of services.

Summary

The text-handling system's architecture is both modular and layered, to enhance its ease of use and flexibility. Its modular design reflects the *model-view-controller* paradigm (originating with Smalltalk-80) where the data, its visual representation, and the logic that links the two are represented by separate objects. In the case of the text-handling system, NSTextStorage holds the model's data, NSTextContainer and NSTextView work together to present the view, and NSLayoutManager intercedes as the controller to make sure that the data and its representation on screen stay in agreement.

This factoring of responsibilities makes each component less dependent on the implementation of the others and makes it easier to replace individual components with improved versions without having to redesign the entire system. To illustrate the independence of the text-handling components, consider some of the operations that are possible using different subsets of the text-handling system:

- Using only an NSTextStorage object, you can search text for specific characters, strings, paragraph styles, and so on.

- Using only an NSTextStorage object you can programmatically operate on the text without incurring the overhead of laying it out for display.

- Using all the components of the text system except for an NSTextView object you can calculate layout information, determining where line breaks occur, the total number of pages, etc.

The layering of the text-handling system reduces the amount you have to learn to accomplish common text-handling tasks. Many applications interact with this system solely through the API of the NSTextView class.

The following sections examine the text-handling system from a practical point of view, showing you how to work with the system to achieve particular goals, starting with the most basic.

Text Input and Output

The text-handling system provides a convenient interface to the file system allowing you to read, display, and write files in these formats:

Format	**Description**
Plain Text	Characters unaccompanied by attribute information.
Rich Text Format (RTF)	Character and attribute information expressed in the Rich Text Format®(RTF). See the *Rich Text Format Specification* by Microsoft Corporation for more information.
Rich Text Format Directory (RTFD)	Character and attribute information expressed in the Rich Text Format but stored in a directory along with the images and other attachments that are embedded in the text.

Text System Defaults

NSModifierFlagMapping (dictionary) (Windows platform only)

This default is on OPENSTEP for Windows only. It allows you to control the mapping between physical modifier keys and logical modifier flags in OpenStep. This default is actually not specific to the text system, but its main purpose is to allow Emacs bindings to work under Windows. By default, both Control keys generate the Command key bit (for menu key equivalents) and both Alt keys generate the Alternate key bit (for mnemonics, primarily). The Control key bit is not available in the default setup which means it is not possible to invoke Emacs-style commands in the text system. This default can be used to remap the available keys to generate what you want. The value of the default is a dictionary with four possible keys, each of which can have one of three possible values. The dictionary keys are: "LeftControl", "RightControl", "LeftAlt", and "RightAlt". The valid values are: "Command", "Alt", and "Control". So the default setup is like this:

```
{
    "LeftControl" = "Command";
    "RightControl" = "Command";
    "LeftAlt" = "Alt";
    "RightAlt" = "Alt";
}
```

One possible setup that allows you to use Emacs keys would be:

```
{
    "LeftControl" = "Command";
    "RightControl" = "Command";
    "LeftAlt" = "Control";
    "RightAlt" = "Alt";
}
```

This would make it so the left Alt key acts like a control key for Emacs. The right Alt key is still used for Alt.

Currently this default has a limitation that only a real Alt key (left or right) can be used for the Alt bit. Therefore it is not valid to assign "LeftControl" = "Alt".

NSMnemonicsWorkInText ("YES" or "NO")

This default controls whether the text system accepts key events with the Alt key down. The default value is NO on Mach and YES on Windows. A value of YES means that any key event with the Alt bit on will be passed up the responder chain to eventually be treated as a mnemonic instead of being accepted by the text as textual input or a key binding command. If this default is set to NO then the key events with the Alt bit set will be passed through the text system's normal key input sequence. This will allow any key bindings involving Alt to work (such as Emacs-style bindings like Alt-f for word forward) and, on Mach it allows typing of special international and Symbol font characters.

NSRepeatCountBinding (key binding style string)

This default controls the numeric argument binding. The default is for numeric arguments not to be supported. If you provide a binding for this default you enable the feature. This allows you to repeat a keyboard command a given number of times. For instance "Control-U 10 Control-F" means move forward ten characters.

NSQuotedKeystrokeBinding (key binding style string)

This default controls the quote binding. The default is for this to be "^q" (that's Control-Q). This is the binding that allows you to literally enter characters that would otherwise be interpreted as commands. For instance "Control-Q Control-F" would insert a Control-F character into the document instead of performing the command **moveForward:**.

NSTextShowsInvisibleCharacters ("YES" or "NO")

The default controls whether a text object will by default show invisible characters like tab, space, and carriage return using some visible glyph. By default it is NO. It only controls the default setting for NSLayoutManagers (which can be modified programmatically). In order for this to work, the rule book generating the glyphs must support the feature. Currently our rule books do not support this feature, so currently this default is not very useful.

NSTextShowsControlCharacters ("YES" or "NO")

The default controls whether a text object will by default show control characters visibly (usually by representing Control-C as "^C" in the text). By default it is NO. It only controls the default setting for NSLayoutManagers (which can be modified programmatically). In order for this to work, the rule book generating the glyphs must support the feature. This feature carries a cost. It will increase the memory needed for documents that contain control characters by quite a lot. Use it with care.

NSTextSelectionColor (color)

This default controls the background color of selected text. By default this is light gray. Kit defaults that accept colors accept them in one of three ways. Either as an archived NSColor object, or as three RGB components, or as a string that can be resolved to a factory selector on NSColor that will return the desired color (for example, "redColor"). Note that NSTextFields and other controls that use field editors to edit their text control their own selection attributes to conform with the platform UI.

NSMarkedTextAttribute and NSMarkedTextColor (color or "underline")

This default controls the way that marked text is displayed. The NSMarkedTextAttributed can either be "Background" or "Underline". If it is "Background" then NSMarkedTextColor indicates the background color to use for marked text. If NSMarkedTextAttribute is "Underline" , NSMarkedTextColor indicates the foreground color to use for marked text (the marked text will be drawn in the specified color and underlined). By default, marked text is drawn with a yellow-ish background color. Kit defaults that accept colors accept them in one of three ways. Either as an archived NSColor object, or as three RGB components, or as a string that can be resolved to a factory selector on NSColor that will return the desired color (for example, "redColor"). For compatibility with the way this default worked in 4.0, if the NSMarkedTextAttribute default contains a color instead of one of the strings "Background" or "Underline" then that color is used as the background color for marked text and the NSMarkedTextColor attribute is ignored.

NSTextKillRingSize (number string)

This default controls the size of the kill ring (as in Emacs Control-Y). The default value is 1 (not really a ring at all, just a single buffer). If you set this to a value larger than one, you also need to rebind Control-Y to "yankAndSelect:" instead of "yank:" for things to work properly (note that **yankAndSelect:** is not listed in any headers). See below for more info on bindings.

Text System Defaults and Key Bindings

OPENSTEP 4.x has a new text system.   This document reveals some tips and tricks about various defaults you can use to customize its behavior. It also describes how to customize the key bindings supported by the new text system.

Note that the new text object exists only in the Release 4.x version of the Application Kit; the following notes don't apply to NEXTSTEP Release 3.3 applications. Also note that the old (3.3) text object exists in the 4.0 Application Kit; these defaults don't apply to it nor to any OpenStep applications which use the old text object.

Heavy-duty subclassers may alter some or all of the text system's functionality, rendering some or all of these features inactive. These notes do apply to NeXT's OPENSTEP applications such as Project Builder, Interface Builder, Text Edit, and others which use the new text system.

The Complete System

The roster of objects that make up the complete text-handling system is relatively long, so this section concentrates on the major players and only mentions the minor ones in passing.

To control layout of text on the screen or printed page, you work with the objects that link the NSTextStorage repository to the NSTextView that displays its contents. These objects are of the NSLayoutManager and NSTextContainer classes.

An NSTextContainer object defines a region where text can be laid out. Typically, an NSTextContainer defines a rectangular area, but by creating a subclass of NSTextContainer you can create other shapes: circles, pentagons, or irregular shapes, for example. NSTextContainer isn't a user-interface object, so it can't display anything or receive events from the keyboard or mouse. It simply describes an area that can be filled with text. Nor does an NSTextContainer store text—that's the job of NSTextStorage.

An NSLayoutManager orchestrates the operation of the other text handling objects. It intercedes in operations that convert the data in an NSTextStorage object to rendered text in an NSTextView's display. It also oversees the layout of text within the areas defined by NSTextContainer objects. To better understand the function of an NSLayoutManager object, you need to understand the difference between characters and glyphs.

The Foundation Framework

Framework: NextLibrary/Frameworks/Foundation.framework

Header File Directories:
NextLibrary/Frameworks/Foundation.framework/Headers

The Foundation Framework defines a base layer of Objective-C classes for OpenStep. In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective-C language. The Foundation Framework 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 OS independence, to enhance portability

The Foundation Framework 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. See "Foundation Framework Classes" for a detailed description of the Foundation Framework.

The Foundation Framework 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 abstract classes like NSEnumerator. These new paradigms reduce the number of special and exceptional cases in API, and allow you to code more efficiently by reusing the same mechanisms with various kinds of objects.

This topic is organized as follows:

The OPENSTEP Text System

Note: *A number of minor changes have occurred to the text system API that may render some explanations, illustrations, and code samples inaccurate. NeXT is working to update this document in a timely manner; a new version should be available from our web site shortly.*

The text-handling component of any application framework presents one of the greatest challenges to framework designers. Even the most basic text-handling system must be relatively sophisticated, allowing for text input, layout, display, editing, copying and pasting, and many other features. But these days developers and users commonly expect even more than these basic features, requiring their simple editors to support multiple fonts, various paragraph styles, embedded images, spell checking, and other features.

A framework that provides these more advanced text-handling features may be adequate for today's programming needs but falls far short when measured against the requirements that are emerging from our ever more interconnected computing world: support for the character sets of the world's living languages, powerful layout capabilities to handle various text directionality and nonrectangular text containers, and sophisticated typesetting capabilities including control of kerning and ligatures.

The OPENSTEP text-handling system is designed to provide all these capabilities without requiring you to learn about or interact with more of the system than is required to meet the needs of your application. It does this by providing a layering of classes, as described in the next section. The sections that follow the architectural overview give you practical examples of how to work with the text-handling system.

This topic is organized as follows:

The Storage Layer: The NSTextStorage Class

An NSTextStorage object serves as the data repository for a group of text handling objects. The format for this data is called an *attributed string*, which is an association of characters (in Unicode encoding) and the attributes (such as font, color, paragraph style) that apply to them. Conceptually, each character in a text has associated with it a dictionary of keys and values. A key names an attribute (say the font) and the associated value specifies the characteristics of that attribute (such as Helvetica 12 point).

An NSTextView lets users affect character attributes through direct action: The user selects some text and reduces the spacing between characters by choosing the Tighten menu command. NSTextStorage lets you operate on the attributes of the text programmatically: Your code can run through the text loosening the kerning for all characters of a certain font and size.

The User-Interface Layer: the NSTextView Class

The vast majority of applications interact with the text-handling system through one class: NSTextView. An NSTextView object provides a rich set of text-handling features and can:

- Display text in various fonts, colors, and paragraph styles
- Display images
- Read text and images from (and write them to) disk or the pasteboard
- Let users control text attributes such as font, super- and subscripting, kerning, and the use of ligatures
- Cooperate with other views to enable scrolling and display of the ruler
- Cooperate with the Font and Spell Check panels.
- Support various key bindings, such as those used in Emacs

The interface that this class declares (and inherits from its superclass NSText) lets you programmatically:

- Control the size of the area in which text is displayed
- Control the editability and selectability of the text
- Select and act on portions of the text

NSTextView objects are used throughout the OPENSTEP user interface to provide standard text input and editing features.

An NSTextView object is a convenient package of the most generally useful text-handling features. If the features of the NSTextView class satisfy your application's requirements, you can skip to the section below titled "Working with the Text-Handling System: Basic Operations". However, if you need more programmatic control over the characters and attributes that make up the text, you'll have to learn something about the object that stores this data, NSTextStorage.

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 twelve string objects will be allocated, no matter how many MonthArray objects exist in an application.

The MonthArray class is declared as:

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

+ sharedMonthArray;
- (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. 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 MonthArray *sharedMonthArray = nil;
static NSString *months[] = { @"January", @"February", @"March",
    @"April", @"May", @"June", @"July", @"August", @"September",
    @"October", @"November", @"December" };

+ monthArray
{
    if (!sharedMonthArray) {
        sharedMonthArray = [[MonthArray alloc] init];
    }
    return sharedMonthArray;
}

- (unsigned)count
{
 return 12;
}

- objectAtIndex:(unsigned)index
{
    if (index >= [self count])
        [NSException raise:NSRangeException format:@"***%s: index
            (%d) beyond bounds (%d)", sel_getName(_cmd), index,
            [self count] - 1];
    else
        return months[index];
}

@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.

Using Services

If you add a Services menu to your application in Interface Builder, there's nothing else you need to do for your application to work with the standard services facility; your application automatically has access to all appropriate services provided by other applications. If you need to construct menus programmatically or if you subclass NSView or NSWindow (or any other subclass of NSResponder), however, you need to do a little work to tie things into the standard services facility. Setting a Services menu programmatically is straightforward. You simply designate the NSMenu that you want as your Services menu with NSApplication's **setServicesMenu:** method. Tying custom NSViews or NSWindows into the standard services facility falls into three steps, in which you invoke or implement these methods:

registerServicesMenuSendTypes:returnTypes:
validRequestorForSendType:returnType:
writeSelectionToPasteboard:types:
readSelectionFromPasteboard:

The following sections cover each of these methods. A final section, "Invoking a Standard Service Programmatically," shows how to invoke a standard service in your code.

Validity of Shared Objects

The Foundation Framework'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 then 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:**.

Variations on Standard Services

The three other types of services—filter, print filter, and spell checker—all share the use of a service specification, but they’re each implemented in different ways. The following sections describe how the service specification for each type of service differs from that for a standard service, and how you take advantage of that type of service.

Working with the Text-Handling System: Basic Operations

The previous section discussed basic operations that can be implemented using the NSTextView and NSTextContainer classes. This section explores those classes in greater depth and brings in the other major classes of the text-handling system, showing you how to use them to accomplish various goals.

Writing Text to a File

Depending on the format of an NSTextView's text, you use slightly different approaches to write the text to a file. For plain text, you extract the contents of the NSTextView as an NSString object and use NSString's **writeToFile:atomically:** method to write the data to disk. RTF text is treated similarly, except that the contents is extracted as an NSData object. Easiest of all is RTFD data, which the NSTextView itself knows how to write to a file:

```
    - (void)saveFile:(id)sender
    {
        NSSavePanel *panel = [NSSavePanel savePanel];


        switch (theFormat) {
          case PlainText:
            [panel setRequiredFileType:@""];
            if ([panel runModal] == NSOKButton) {
                [[theTextView string] writeToFile:[panel filename]
                    atomically:YES];
            }
            break;

          case RichText:
            [panel setRequiredFileType:@"rtf"];
            if ([panel runModal] == NSOKButton) {
                [[theTextView RTFFromRange:NSMakeRange(0, [[theTextView string]
                    length])] writeToFile:[panel filename] atomically:YES];
     }
     break;

   case RTFD:
     [panel setRequiredFileType:@"rtfd"];
     if ([panel runModal] == NSOKButton) {
         [theTextView writeRTFDToFile:[panel filename] atomically:YES];
     }
     break;

   default:
     NSRunAlertPanel(@"Save Error",

         @"Couldn't save file (unknown data format).\n", nil, nil, nil);
     break;
  }
  return;
}
```