9 *User-Interface Objects*

This information, although fundamentally correct, has not been updated for release 3.0. For up-to-date information on the Text and Box classes, see the documentation in:

/NextLibrary/Documentation/NextDev/GeneralRef/02_ApplicationKit/Classes

---

The Application Kit consists mainly of class definitions for objects that respond to events or draw on the screen. Chapter 6, ªProgram Structure,º and Chapter 7, ªProgram Dynamics,º showed how these objects work together to ground your application in the NeXT window system and give it a unified program structure. Chapter 8, ªInterface Builder,º showed how to build a program from these objects graphically, writing only minimal amounts of Objective-C code.

This chapter discusses each of these objects in more detail, from the standpoint of the Application Kit, not Interface Builder. This information will be useful if you want to gain a full understanding of how these objects work, if you plan to define a subclass of any of them to change default behavior, or if you simply want to know the range of possibilities allowed by their kit definitions.

All user-interface objects are Responders, and most are also Views of one sort or another. They fall roughly into five categories:

- The Application object that oversees the entire program. You'd generally customize this object to meet the needs of your program by defining an Application subclass.
- Control objects, Views that implement the target-action paradigm and give users a way to control the program's activity.
- Window objects that manage the windows provided by the Window Server and the Views that are displayed within them.
- The Text object, which gives programs a way to display text to users and users a way to enter, edit, and select text.
- Views that are generally used in conjunction with other Views, either to provide scrolling and zooming support, or to provide a background and frame to a companion View.

For the most part, these are objects that you can use ªoff the shelf.º You get an object that's ready to use just by creating and initializing an instance of a Kit class; you don't have to define your own subclass.

The Text Class

The Text class is a subclass of View designed for the display and manipulation of text. An object of the Text class can:

- Display text in various fonts.
- Control the format of lines and paragraphs.
- Wrap lines within the boundaries you set.
- Specify displayed text as either read-only or editable.
- Let the user cut, copy, and paste text between windows in the same or different applications.
- Write text to, or retrieve it from, the disk.

A Text object can be used to implement notepads, static text displays, and electronic forms, to name a few possibilities.

Many applications will include a Text object only indirectly, through the use of a TextField or Form object. TextField and Form objects provide a simplified interface to the Text object. They initialize the Text object for you and convert its contents to the format you request: an integer, floating-point, or string value. Since it belongs to a subclass of Control, a TextField or Form object also lets you set an action message and a target. Furthermore, all TextField and Form objects in the same window share a single Text object, thus reducing the memory demands of your application.

TextField and Form give you access to a small but frequently used subset of the methods that the Text class implements. If the text-handling features provided by these Controls don't meet your needs, use a Text object directly. Because the Text class has a rich selection of features, you'll rarely need to create a Text subclass of your own.

Operation

This section gives a general overview of how a Text object works. Keep in mind that through its many methods, a Text object's standard behavior can be altered in fundamental ways.

A Text object is a View specifically designed for text display and editing. It draws in a flipped coordinate system; that is, the origin is located at the bounds rectangle's upper left corner. x-axis values increase from left to right and y-axis values increase from top to bottom. Thus, x values increase for each new character on a line, and y values increase for each new line. See Chapter 6 for general information on View coordinate systems.

Being a View, a Text object renders on the screen only the drawing that lies within its frame rectangle. However, the drawing of characters is confined to a generally smaller area, the *body rectangle*. This rectangle is inset from the edges of the frame rectangle by the width of the Text object's margins, as shown in Figure 9-1. Between the body rectangle and the frame rectangle, a Text object draws only its background color. Within the body rectangle, characters are drawn in horizontal lines that stretch from the left to the right margin. The height of each line is calculated to accommodate the tallest character in the line.

F0.eps ,

Figure 9-1. Text Layout

A Text object views its contents as a string of characters. Individual characters are identified by their *character position*, their numerical rank in the string starting from 0. For example, if in your application you need to select a group of characters programmatically, you must specify their starting and ending positions.

As a Text object draws a line of text, it continually recalculates the accumulated width of all the characters drawn so far on that line. This calculation takes into account the width of each character in the line (given its font) as well as the width of any tabs or paragraph indentations. If the total width exceeds that available between the left and right margins, a portion of the text is wrapped onto the next line. Lines of text are generally wrapped on a word basis.

By default, the text a Text object displays can be edited by the user. However, there are two mechanisms you can use to protect the text from alteration. One lets you specify that the text is read-only, and the other lets another object within your program decide dynamically when the text can be edited and when it can't. This object is the Text object's delegate.

Besides determining whether the text can be edited, a Text object's delegate can control whether the Text object can resize its frame rectangle in response to the addition or deletion of text and whether it can accept or resign the status of first responder. After allowing any of these changes, the delegate is notified of the Text object's new state.

If its text is editable and the Text object becomes its window's first responder, it receives key events. As key events arrive in the Text object, a *character filter function* examines each character to determine the appropriate action. It passes some characters into the text and interprets others as a signal that the user is finished entering text. By default, Tab and Return characters are accepted into the text, although you can easily configure a Text object to interpret them as a signal to end the editing session.

You can also specify that incoming text, whether from the keyboard, the pasteboard, or a file, is passed to a *text filter function*. This filter is similar to the character filter but more versatile since it's passed more information

about the state of the text, the insertion point, and the Text object. By using this added information, a text filter can implement auto-indentation or other automatic formatting features.

The sections that follow describe these concepts in more detail.

Creating a Text Object

You create a Text object by sending a **newFrame:text:alignment:** message to the Text class:

```
myText =[Text newFrame:&aRect text:"Brevity is the soul of wit."
                alignment:NX_LEFTALIGNED];
```

The Text class responds by creating a Text object. The object's frame rectangle, text, and text alignment are set by the method's three arguments:

Argument	**Permitted Value**
frame	A pointer to an NXRect structure. This structure specifies the frame's size and the location of its lower left corner within the coordinate system of the Text object's superview.
text	A pointer to a null-terminated array of characters. These characters appear when the Text object is displayed.
alignment	A constant specifying the alignment of the text. Text can be aligned with the left or right margin or each line can be centered between the left and right margins. The choices are: NX_LEFTALIGNED NX_RIGHTALIGNED NX_CENTERED

You can also create Text objects by sending **new** or **newFrame:** messages:

```
myText1 = [Text new];
myText2 = [Text newFrame:&aRect];
```

The **new** method sends a message to the Text class to apply the **newFrame:text:alignment:** method. The arguments it sends are a frame rectangle of 0, a null character pointer, and left text alignment. You could then send **sizeTo::** and **setText:** messages to the Text object that's created to set its size and text. (These instance methods are discussed in more detail in the following sections.)

The **newFrame:** method is similar to the **new** method except that it lets you specify the size and location of the Text object's frame rectangle.

The Text object you create will use the default system font. By sending the appropriate message to the Text class, you can specify a different default:

```
[Text setDefaultFont:anObject];
myText = [Text newFrame:&aRect];
```

If you use the **setDefaultFont:** method, be sure that the y-axis of the specified font has the same orientation as that of the Text object. Since the Text object flips the orientation of its y-axis, the font must also have a flipped y-axis to prevent characters from being drawn upside-down:

```
[Text setDefaultFont: [Font newFont:"Times-Roman" size:12.0 style:0
                        matrix:NX_FLIPPEDMATRIX]];
myText = [Text newFrame:&aRect];
```

If you use NULL as the argument to **setDefaultFont:**, the Text object will use the default system font, as defined by the defaults system.

The Text

A Text object provides methods that let you control the form and content of the text. Other methods let you specify whether the user can add, alter, or delete a Text object's text. This section discusses how you can specify what text a Text object displays; the section ªText Editingº discusses the subjects of read-only and editable text.

Setting the Text

The text the Text object displays can be set in a number of ways. It can be:

- Set at the time the object is created
- Set by a message from another object in the application
- Read into the Text object from a file
- Entered by the user

The previous section, ªCreating a Text Object,º gives an example of setting a Text object's text at the time the object is created. After a Text object has been created, another object in the application can reset the text by sending a **setText:** message:

```
[myText setText:"O, for a horse with wings!"];
```

The argument is a pointer to the array of characters that will appear in the Text object. Any text the Text object previously contained will be replaced by this string. To append text to the contents of a Text object, you have to manipulate the selection, as described in the section ªThe Selectionº below.

You can also set the Text object's text by applying the **readText:** or **readRichText:** method. These methods set the text to the contents of the stream you specify. The stream can represent a Mach port, a portion of memory, or, as in this example, a file.

```
NXStream  *input;
char      *theFile = "Sonnets";

input = NXMapFile(theFile, NX_READONLY);
[textOutlet readText:input];
NXCloseMemory(input, NX_FREEBUFFER);
return self;
```

As before, the Text object's previous text is replaced by the new text.

readText: interprets the data in the stream as plain ASCII text. The text is displayed using the Text object's default font and layout characteristics.

A similar method, **readRichText:**, is designed to read text encoded in Rich Text Format®, or RTF, a format for storing text and graphics. (For more information on RTF, see the *Rich Text Format Specification* by Microsoft

Corporation.) The Text object uses the font and layout information embedded in the RTF file as a guideline for displaying the text. If a Text object doesn't support a specific RTF formatting instruction, it simply ignores it; no error occurs. See the ªRich Text Format Supportº section below for a table of supported RTF control words.

Finally, the user can set the text in the Text object. See ªText Editingº below for more information.

Examining the Text

Your application can get a copy of all or part of the Text object's text by sending a **getSubstring:start:length:** message. This method takes three arguments: a pointer to a character array, the starting position of the substring, and the total number of characters to copy into the array. Substring positions are relative to the first character of the text, which is taken to be at position 0.

If you want to copy the Text object's entire text, pick a value for **length** that equals or exceeds the actual text length. No error results if **start** plus **length** specifies a character position beyond the end of the text. For example, to copy the entire contents of **myText**, first use the **textLength** method to determine the text's length. Then, send a **getSubstring:start:length:** message with that value:

```
char  *textBuffer;
int    textLength;

textBuffer = malloc([text textLength]+1);
[text getSubstring:textBuffer start:0 length:[text textLength]];
```

If the string you request encompasses the entire text, the Text object appends a null terminator ('\0') to the string. As the example illustrates, you should allow for the null terminator when you allocate storage for the string. If you only want a portion of the entire text, you'll have to add the null terminator yourself. This code fragment copies a null-terminated string containing the first 100 characters of **myText**:

```
char  textBuffer[101];

[myText getSubstring:textBuffer start:0 length:100];
textBuffer[100] = '\0';
```

getSubstring:start:length: returns an integer indicating the actual number of characters copied. The total doesn't include the null terminator that **getSubstring:start:length:** adds if the requested string includes the

entire text. A returned value of -1 indicates that the starting position is beyond the end of the text.

Writing the Text to a File

A Text object can write its text out in either ASCII or RTF format. ASCII format retains only minimal information about the original text. It records the text's characters but no information about fonts, sizes, paragraph indentation, or tab settings. In contrast, Rich Text Format encodes enough information to ensure that the text could be restored to its original specifications. The Text class currently supports only a subset of the formatting commands defined in the RTF specification. See the ªRich Text Format Supportº section below for a table of supported RTF control words.

Note: A Text object writes to a stream rather than to a file. (See the ªApplication Kit Conventionsº section of Chapter 6 for information on streams.) The examples that follow assume you want to store the data from the stream into a file.

To write an ASCII version of a Text object's text to a file, use the **writeText:** method:

```
int         fd;
NXStream   *stream;
char       *theFile = "Sonnets";

fd = open(theFile, O_CREAT | O_WRONLY | O_TRUNC, 0666);
stream = NXOpenFile(fd, NX_WRITEONLY);
[myText writeText:stream];
NXFlush(stream);
NXClose(stream);
close(fd);
```

To write only a portion of the text, use **getSubstring:start:length:** to get a copy of the text and then **NXWrite()** to write the string to the stream:

```
int         fd;
NXStream   *stream;
char       *theFile = "Snippet";
char        textBuffer[11];

fd = open(theFile, O_CREAT | O_WRONLY | O_TRUNC, 0666);
```

```
stream = NXOpenFile(fd, NX_WRITEONLY);
[textOutlet getSubstring:textBuffer start:0 length:10];
textBuffer[10] = '\0';
NXWrite(stream, textBuffer, 10);
NXFlush(stream);
NXClose(stream);
close(fd);
return self;
```

See ªThe Selectionº below for information on writing only the selected text to the disk.

Text Layout

The format of a Text object's displayed text depends on a number of factors, including:

- The size of the Text object's frame rectangle
- The dimensions of the four margins
- The alignment of the text
- The vertical placement of the characters within the line
- The style of each paragraph

Frame Rectangle

As described in ªCreating a Text Objectº above, you can set the size and location of the frame rectangle when you create a new Text object. A Text object can also dynamically change the size of its frame rectangle in response to the addition or deletion of text. The simplest way to control resizing is with the **setVertResizable:** and **setHorizResizable:** methods. (You can also let the Text object's delegate control resizing; see ªThe Delegateº below.) These methods take a boolean value that determines the dimension, width or height, that the rectangle can alter.

```
[myText setVertResizable:YES];
[myText setHorizResizable:NO];
```

The **isVertResizable** and **isHorizResizable** methods report the resizing status of a Text object.

Assuming the Text object can be resized, the **setMaxSize:** and **setMinSize:** methods set limits to the extent of the change:

```
NXSize maxSize = {100.0, 10000.0};
NXSize minSize = {100.0, 20.0};

[myText setMaxSize:&maxSize];
[myText setMinSize:&minSize];
```

This example allows a Text object's frame rectangle to grow to the specified maximum height as text is added or to shrink to the specified minimum size as text is deleted. Use the **getMaxSize:** and **getMinSize:** method to learn the current settings of these size limits.

If a Text object is resizable, sending a **sizeTo::** message will resize it within the constraints set by the four methods introduced above. Note that **sizeTo::** doesn't recalculate the placement of text within the newly resized frame rectangle. For this reason, you must send a **calcLine** message immediately after resizing a Text object.

A **sizeToFit** message resizes a Text object so that all of the text it contains is displayed. Again, this resizing is subject to the limits set by the four methods above. Use **calcLine** to rewrap the text after sending a **sizeToFit** message.

Margins

Text layout is also determined by the width of a Text object's margins. The margins determine the amount the body rectangle is inset from the sides of a Text object's frame rectangle. By default, the body rectangle's height is the same as that of the frame rectangle, but its width is 4.0 units (in the Text object's current coordinate system) narrower than the width of the frame rectangle. The body rectangle is centered between the left and right edges of the frame rectangle.

The **getMarginLeft:right:top:bottom:** method reports the current settings of the four margins. To alter these values, send a **setMarginLeft:right:top:bottom:** message. With this method, you can reset all four values at once. You might use both methods if you wanted to reset only one margin. For example, to reset the top margin to 6.0 while leaving the other margins at their current settings, you could send these messages:

```
NXCoord  lMgn, rMgn, tMgn, bMgn;
```

```
[myText getMarginLeft:&lMgn right:&rMgn top:&tMgn bottom:&bMgn];
[myText setMarginLeft:lMgn right:rMgn top:6.0 bottom:bMgn];
```

Alignment

Text alignment also affects the layout. By default, text is aligned with the left margin, which causes a block of text to have a ªragged rightº edge. After a Text object is created, its text alignment can be queried by sending an **alignment** message and can be reset by sending a **setAlignment:** message.

The **setAlignment:** method uses the same constants (NX_LEFTALIGNED, NX_RIGHTALIGNED, and NX_CENTERED) as the **newFrame:text:alignment:** class method described in ªCreating a Text Objectº above. As the names of the constants suggest, text can be aligned to the left or right margin or centered between the left and right margins.

Sending a **setAlignment:** message doesn't redraw the text; you must redisplay the text to exhibit the change. For example:

```
if ([myText alignment] != NX_CENTERED) {
    [myText setAlignment:NX_CENTERED];
    [myText display];
}
```

Line and Character Layout

Methods defined in the Text class let you adjust the vertical placement of characters within a line as well as the height of the line itself, giving you the ability to display single-spaced or double-spaced text, or any other spacing you choose. The default values for line height and character placement depend on the Text object's font, but are designed to give single-spaced text for the font that's currently in use.

In the context of the Text class, a line is a rectangular area that extends from the left to the right edge of the Text object's body rectangle and that contains a single row of characters. Each line within a paragraph shares a side with the preceding and the following line; thus setting the line spacing for a Text object is, in fact, a matter of setting the height of the lines.

Within a line, a Text object positions a character according to two measurements. It sums the widths of the

preceding characters to determine the x-coordinate, and it sets the y-coordinate a certain distance above the bottom of the line's rectangular drawing area. Figure 9-2 illustrates character placement in a line of text.

F1.eps ,

Figure 9-2. Line and Character Metrics

Characters within a font are horizontally aligned relative to a reference line called the *baseline.* You can think of characters such as ªxº and ªhº as standing on the baseline. For lowercase letters, the vertical extent of a letter is divided into three parts: the ascender, the descender, and a middle section. The middle section extends from the baseline to the top of an ªxº character, a height known as the *x-height*. The portion of a character that extends above the x-height is called the *ascender*; the portion below is called the *descender*. The distance from the baseline to the bottom of the line's drawing area is called the *descent line*.

The Text class's **setLineHeight:** and **lineHeight** methods set and report a Text object's line height. To display double-spaced lines of text, you would reset the Text object's line height. Assuming the Text object's text is currently single-spaced, you could specify double-spaced text by sending these messages:

```
[myText setLineHeight:2.0 * [myText lineHeight]];
```

To adjust the vertical placement of characters within a line, you can change the descent line. The Text object's **descentLine** and **setDescentLine:** report and set the descent line. For example, to double the current value of the descent line:

```
[myText setDescentLine:2.0 * [myText descentLine]];
```

Although you might specify a line height or descent line value that would cause a character to extend outside of a line's rectangular drawing area, a Text object will override these values so that even the largest character on the line is fully visible.

Word and Character Wrapping

A Text object continues to draw characters on a single line until adding another character would cause the total width of the characters to exceed the space between the line's right and left margins. What happens next depends on the previous characters on the line and the current state of the Text object:

- The new character, and the word it's part of, can appear on the next line: The text is *word-wrapped*.
- The new character alone can appear on the next line: The text is *character-wrapped*.
- The new character can be added to the Text object's text but not be drawn within the frame rectangle: The drawing of the new character, and any that follow it on the same line, is clipped by the Text object's body rectangle.
- The width of the Text object's frame rectangle can grow to accommodate the new character.

A Text object's default behavior is to word-wrap lines of text. If adding text to a line would push some character on the line outside the body rectangle, the Text object takes the word this character is part of and redraws the entire word on the next line. Unlike a UNIX text editor such as Emacs, a Text object wraps the text without adding a new line character ('\n') at the end of each line. Wrapping affects only the display of the text, not its contents. Consequently, if you change the width of a Text object's body rectangle, the text it contains will be rewrapped to the new line length.

To determine word boundaries, a Text object refers to a *break table*. Although it's unlikely you'll ever need to change the standard break table, the Text class provides the **setBreakTable:** and **breakTable** methods to give you that option. See ªText Tables and Filter Functionsº below for more information on the break table.

If the Text object can't find a word boundary on the current line (that is, a single ªwordº spans the body rectangle's width), it wraps new characters onto the next line. The **setCharWrap:** and **charWrap** methods let you control how a Text object treats such long words:

```
if([myText charWrap] == YES)
    [myText setCharWrap:NO];
```

After receiving a **setCharWrap:**NO message, a Text object's word wrap feature is still enabled, but the display of long words is clipped to the edge of the body rectangle. Although the clipped characters aren't visible, they're still part of the Text object's text. By deleting characters that precede them on the line, or by entering Return in the midst of the long word, the user can bring the clipped characters into view.

You can disable both word-wrapping and character-wrapping by sending a **setNoWrap** message:

```
[myText setNoWrap];
```

After a Text object receives a **setNoWrap** message, a user can begin a new line only by entering Return. A Text object allows no more than 16,384 (or 2_{14}) characters on a single line.

If the Text object is resizable, it can grow to accommodate text that would exceed the width of the body rectangle. If the enlarged frame rectangle still falls within its superview's frame rectangle, the new text will be visible. See the ªFrame Rectangleº section above and ªThe Delegate,º below, for more information.

The Selection

The selection is the portion of the text that an action, such as a command, will affect. The selection can have 0 width (an *empty selection*), in which case it's indicated by a blinking vertical bar called the *caret*, or it can contain one or more characters. If the selection spans one or more characters, it's indicated by highlighting. A Text object automatically chooses the proper marking depending on the length of the selection.

Highlighting is only an attribute of the selection; it shouldn't be confused with the selection itself. By setting the selection's gray value to be equal to that of the background (see ªDrawing and the Text Classº below), you can easily create a selection that doesn't display a visible highlight.

By default, a Text object's text is selectable. The **isSelectable** method reports whether the text is selectable, and the **setSelectable:** method lets you alter this status:

```
if ([myText isSelectable] == YES)
    [myText setSelectable:NO];
```

Only selectable text is editable; however, selectability doesn't necessarily imply editability. You can have selectable, read-only text. For example, you may want the user to select some text, say the answer to a multiple-choice question, but not be able to alter that text. The ªText Editingº section that follows describes how to protect text from alteration.

The Text class provides methods that affect a selection's length, character contents, font, gray value, and

location in a scrolling view. A selection's font and gray value are discussed later under ªThe Fontº and ªDrawing and the Text Classº; the other concepts are introduced in the next two sections.

Setting the Selection

The selection can be set by the user or it can be set programmatically. In most cases, the selection is set by the user's actions with the mouse, by clicking, dragging, or a combination of the two. See Chapter 2, ªThe NeXT User Interface,º for a general discussion of the semantics of mouse actions. The rest of this section discusses how to manage the selection programmatically.

The visibility of the caret, which marks the empty selection, is generally determined by the Application Kit itself rather than by the code you write. When a Text object becomes the first responder, it sends itself a **showCaret** message to start the timed entry that blinks the caret. Conversely, before a Text object stops being the first responder, it sends itself a **hideCaret** message to erase the caret from the text and to remove the timed entry. You'll rarely need to send these messages in your own code.

The position and extent of the selection can be set programmatically using the **setSel::** method. This method takes two integer arguments corresponding to the first and last character positions of the selection. If the two arguments are identical, **setSel::** has the effect of setting the position of the caret.

```
[myText setSel:0:0];
```

In this case, the caret moves to the start of the text. Note that the selection is measured from the left edge of the character at the specified position. Character position 0 is the position of the first character of the text; sending a **setSel:**0 **:**0 message moves the caret immediately to the left of the first character.

To select the first ten characters of the text, send this message:

```
[myText setSel:0 :10];
```

Since the selection is measured from the left edge of the character position, this message selects the characters at character position 0 through 9.

The Text class provides the **selectText:**, **selectAll:**, and **selectError** methods for selecting the entire text. **selectText:** and **selectAll:** are synonyms. They attempt to make the text object the first responder and then, if successful, select the entire text. The **selectError** method is similar to these other two except that it doesn't

attempt to alter the Text object's responder status before selecting the text. These methods are used primarily in support of the Text class's delegate/notification system, as described in ªThe Delegateº below.

Querying the Selection

The **getSel::** method returns the starting and ending positions of the current selection, along with other information. Instead of taking **int** arguments like **setSel::**, **getSel::** takes pointers to **NXSelPt** structures. An **NXSelPt** structure has these fields:

```
typedef struct _NXSelPt {
    int      cp;    /* the character position */
    int      line;  /* line offset */
    NXCoord  x;     /* x coordinate of character position */
    NXCoord  y;     /* y coordinate of character position */
    int      c1st; /* character position of first character in line */
    NXCoord  ht;    /* line height */
} NXSelPt;
```

The first field, **cp**, gives the character position of one end of the selection. The value of **line**, interpreted in light of information that a Text object keeps in an internal table of line breaks (see ªThe Break Tableº in ªText Tables and Filter Functionsº below), identifies the line of text containing this character position. The next two fields give the x and y location of the top corner of this end of the selection. **c1st** identifies the character position of the first character of the line containing **cp**. The last field, **ht**, gives the height of the line containing **cp**. Two **NXSelPt** structures give a Text object the information it needs to identify and highlight the selected characters.

Using the character position information from the **NXSelPt** structure, you could copy the selected text to another buffer:

```
NXSelPt  selStart, selEnd;
char     buf[100];
int      selLength;

[myText getSel:&selStart :&selEnd];
selLength = selEnd.cp - selStart.cp;
[myText getSubstring:buf start:selStart.cp length:selLength];
buf[selLength+1] = '\0';
```

The **replaceSel:** method lets you alter the contents of the selection. With this method, for example, your application could replace a selected misspelled word with the correct spelling. You can also use **replaceSel:** to append text to the current contents of a Text object:

```
int      length;

length = [myText textLength];
[myText setSel:length :length];
[myText replaceSel:"\n\nYours truly,\n"];
```

Text Editing

The Text class provides two ways to control the editability of a Text object's text; the approach you take will depend on your needs. To set the editing status for the duration of the program's execution, use the **setEditable:** method. To allow editing under some circumstances but not others, let the Text object's delegate determine if and when the text is editable.

Note that these two systems can't be used at the same time. Sending a **setEditable:** message disables the delegate/notification system.

Setting Editability

The **setEditable:** method lets you specify the editability of the text, and the **isEditable** method reports the text's current editing status:

```
if([myText isEditable] == YES);
    [myText setEditable:NO];
```

The following sections describe how a delegate can control the editability of the text.

The Delegate

A Text object can send notification messages to a delegate in response to user actions. The delegate's response determines the Text object's behavior, such as whether it will allow the user to edit its text.

You set and query a Text object's delegate just as you would a Window's:

```
[myText setDelegate:anObject];
anObject = [myText delegate];
```

A Text object sends predefined messages to its delegate depending on the user's specific input. These messages are described in the next section.

When you set the delegate, the Text object queries it to discover which of the predefined messages it responds to. Thereafter, the Text object only sends those messages that the delegate can accommodate. As noted below, a Text object reverts to its default behavior if its delegate doesn't respond to a particular message, or if there is no delegate object.

If, in your application, more than one Text object reports to the same delegate, you'll need a way to identify which text object sent the notification message. See ªThe Tagº below for such an identification system.

Notification Messages

The notification messages a Text object sends to its delegate can be divided into two categories: those that are sent before a change is made and those that are sent after. Messages in the first group advise the delegate of an impending change, giving the delegate the chance to allow or prevent the change. Messages in the second group simply report that a change has been made.

Before Change	After Change
textWillChange:	textDidChange: text:isEmpty:
textWillEnd:	textDidEnd:endChar:
textWillResize:	textDidResize:oldBounds:invalid:

The textWillChange:, textDidChange:, and text:isEmpty: Messages

The delegate receives a **textWillChange:** message the first time the user attempts to change the text after a Text

object has become the first responder. Only one **textWillChange:** message, the one accompanying the first attempted change, is sent during the time the Text object retains first responder status. The delegate, depending on the value returned by its **textWillChange:** method, either allows or prevents the change.

If the delegate's **textWillChange:** method returns NO, the change is allowed; if it returns YES, the change is prevented. You can think of a NO return value as indicating that the delegate has ªno objectionº to changes in the Text object.

```
- (BOOL) textWillChange:(id)textObject
    {
        return(NO);
    }
```

If the delegate doesn't implement the **textWillChange:** method, or if the Text object has no delegate, the change is allowed by default.

If the delegate's response to the **textWillChange:** message allows the change, the delegate immediately receives a **textDidChange:** message. Thereafter, each time the user changes the text either by using the keyboard or the pasteboard, the delegate receives a **text:isEmpty:** message. This message passes the Text object's **id** and a boolean value indicating whether the Text object contains any text after the change. One example of the use of this information is the Save panel. The OK button is enabled or disabled depending on whether there's text in the text field provided for the file name.

When a user is entering characters from the keyboard, the Text object attempts to send a **text:isEmpty:** message for each character. A fast typist might outrun this system momentarily, but it's unlikely such a typist could enter more than two or three characters before the Text object sends a message. When a user cuts text from or pastes text into a Text object from the pasteboard, the Text object sends a single **text:isEmpty:** message to alert the delegate of the change.

The textWillEnd: and textDidEnd:endChar: Messages

Before a Text object ceases to be the first responder, it sends a **textWillEnd:** message to its delegate. The delegate can either allow the change or prevent it, depending on the return value of its **textWillEnd:** method.

If the delegate objects to the change, its **textWillEnd:** method returns YES; otherwise, it returns NO. If the delegate objects, the Text object remains the Window's first responder and selects the entire text (see ªThe

Selectionº below).

The **textWillEnd:** message gives the delegate a chance to validate the Text object's text before letting the user select some other object:

```
- (BOOL) textWillEnd:(id)textObject
{
    char  *modelBuffer = "The accepted answer.";
    char  *textBuffer;

    /* get entire text */
    textBuffer = malloc([text textLength]+1);
    [myText getSubstring:textBuffer start:0
        length:[myText textLength]+1];

    /* compare the text to the model */
    if (!strcmp(textBuffer, modelBuffer))
        /* they're the same */
        return(NO);
    else
        /* they're different */
        return(YES);
}
```

If the delegate doesn't implement the **textWillEnd:** method, or if the Text object has no delegate, the change is allowed by default.

After a Text object ceases to be the first responder, it sends a **textDidEnd:endChar:** message to its delegate. The two arguments passed with this message are the Text object's **id** and the character that caused the Text object to stop being the first responder. The delegate can use this information to decide which other object should become the first responder. For example, if the character was Tab, the delegate would typically change the first responder to the next Text object in the window.

The Text object's character filter determines which characters the Text object interprets as a command to cease being the first responder. See ªCharacter Filter Functionsº below for more information.

The textWillResize: and textDidResize:oldBounds:invalid: Messages

The Text object's bounds rectangle can change size to accommodate a change in the amount of text. The bounds rectangle can either grow to hold an increased amount of text or shrink if some of the text is deleted. In either case, before the change can take place, the Text object sends a **textWillResize:** message to its delegate, passing its **id** as the message's argument.

As with the **textWillChange:** and **textWillEnd:** methods, the delegate's **textWillResize:** method can prevent the bounds rectangle from changing size by returning YES. If, on the other hand, the delegate permits the Text object's bounds to change size, it can specify to what degree and in which directions the change can occur:

```
- (BOOL) textWillResize:(id)textObject
{
    NXSize maxSize = {100.0, 10000.0};
    NXSize minSize = {100.0, 100.0};

    [myText setMaxSize:&maxSize];
    [myText setMinSize:&minSize];
    return NO;
}
```

This **textWillResize:** method lets the Text object's bounds rectangle grow to 10000.0 units in the Text object's coordinate system in the vertical (or y) direction. No growth is allowed in the horizontal (or x) direction. It's not permitted to change size in either direction unless the Text object has been configured to be resizable as described in ªFrame Rectangleº above.

If the delegate doesn't implement this method, or if the Text object has no delegate, the change is allowed by default.

The Text object sends a **textDidResize:oldBounds:invalid:** message to its delegate after the bounds rectangle's size has been changed. The Text object passes its **id** as the first argument and its old bounds rectangle as the second argument. The third argument is the area of the Text object's superview that should be redrawn if the Text object's new bounds rectangle is smaller than its old bounds rectangle.

The Tag

You can assign a tag, an arbitrary integer, to a Text object as a means of identifying it to its delegate:

```
[myText1 setTag:1];
[myText2 setTag:2];
```

A tag gives the delegate a way to determine which Text object, among several, has sent a particular notification message:

```
- (BOOL) textWillChange:(id)textObject
{
    if([textObject getTag] == 1)
        /* prevent the change */
        return(YES);
    else if([textObject getTag] == 2)
        /* allow the change */
        return(NO);
}
```

Smart Cut and Paste

A Text object attempts to respect word spacing conventions when the user cuts words from or pastes them into the text. For example, consider this sentence:

ªI see no objection to stoutness, in moderation.º
ÐW.S. Gilbert 1836-1911

If the user double-clicks the word ªnoº and then cuts it, the word and space character to its left disappear. This leaves only one space between the words that had flanked ªnoº. Furthermore, if the user double-clicks the word ªmoderationº and then cuts it, the period is brought up tight to the word ªinº. Similarly, a word pasted into a line of text adds a space character when needed. For example, pasting the word ªstrongº between the words ªno objectionº results in ªno strong objectionº. It doesn't matter if the insertion point is to the left or right of the intervening space, standard word spacing is preserved. Pasting a word between an existing word and its following punctuation likewise leaves no unwanted spaces. Note that a Text object only uses ªsmartº cut and paste on word selections. The user must double-click, double-click and then Shift-click to extend the word selection, or double-click and drag to select a series of words to invoke this feature of the Text class.

See ªThe Smart Cut and Paste Tablesº below for details on how this feature is implemented.

Cut, Copy, Paste, and Delete

The Cut, Copy, Paste, and Delete commands of a standard application's Edit menu send, respectively, **cut:**, **copy:**, **paste:**, and **delete:** messages to the first responder. If you create an application using Interface Builder, these messages are already assigned to the appropriate menu item. A Text object is typically a Window's first responder and so must interpret these messages as editing commands.

Note, however, that not all Text objects may become first responder. To become first responder, a Text object must be selectable and, if it has a delegate, the delegate must allow the change of status. If a Text object doesn't meet these criteria, these editing commands will have no effect on it.

The simplest of these four editing methods is **delete:**. Assuming the current selection includes one or more characters, a **delete:** message removes the selected text from the Text object. The Text object rewraps and redisplays the remaining text and replaces the extended selection with the caret. If there is a delegate, and it implements the method, the Text object sends it a **text:isEmpty:** message. A **delete:** message affects only the text: No text is placed on the pasteboard.

copy: is the complement of **delete:**. A **copy:** message affects only the contents of the pasteboard, leaving the text unaltered. Again assuming a selection of positive length, a **copy:** message opens the pasteboard and writes the selected text to it. The Text object puts three types of data relating to the selection on the pasteboard:

- The text
- Format information
- Selection-type information

The text is simply an ASCII string of a specified length. Format information consists of the font characteristics and paragraph styles included in the copied text. The selection type records whether the text was the result of word selection. A Text object that receives this text can use the selection-type information to determine correct word spacing for the inserted text. Since a **copy:** message doesn't alter the text, no notification message is sent to the Text object's delegate.

The **cut:** method combines the functions of the **copy:** and **delete:** methods. In fact, when a Text object receives a **cut:** message, it sends itself a **copy:** message to copy the selected text to the pasteboard and then sends itself a **delete:** message to remove the selected text.

Assuming a Text object contains editable text, a **paste:** message causes it to read the contents of the pasteboard and insert the text. If the current selection is marked by a caret, the text is inserted at that point. If the current selection includes one or more characters, the inserted text replaces these characters. In either case, the Text object places the new insertion point after the inserted text.

Before inserting the pasteboard's contents into the text, a Text object checks whether this text was the result of word selection. If so, the Text object refers to its smart cut and paste tables to determine whether a space character should be added to one or both ends of the string to maintain proper word spacing around the inserted text.

The pasteboard can also contain formatting information for the text it contains. Depending on the Text object's state, it can either preserve the inserted text's previous format or ignore it and make the new text conform to the Text object's prevailing format. By default, a Text object ignores the formatting information from the pasteboard. The **setMonoFont:** method, as described in ªThe Text's Fontº below, controls how a Text object handles formatting information from the pasteboard.

If the Text object's delegate implements the corresponding method, it receives a **text:isEmpty** message when text is pasted into the Text object.

The Font

The Text class defines methods for setting the name, size, and style of the font for the entire text or any portion of it. A Text object alters the characteristics of a font by sending messages to a Font object. Although you can set a Text object's Font object directly, it's generally easier to use Text methods that let you set the individual font characteristics. The two sections below introduce the methods that affect a Text object's fonts.

A Text object actually receives two fonts for each font it requests. These two fonts are identical except for character width information: One font is designed for screen use and the other for the printer. In determining line breaks for some text, a Text object makes two calculations, one using screen widths and the other using printer widths. The Text object compares the results of these calculation and then breaks each line based on the result of the calculation that leaves the least text on a line. In this way, when comparing printed output to text on the screen, line breaks are consistent although precise line lengths may differ.

The Text's Font

You can set the default font for a Text object using **setDefaultFont:**, as discussed in ªCreating a Text Objectº above. Once a Text object is created, use the **setFont:** method to set the font of the entire text. This method takes a Font **id** as its sole argument:

```
[myText setFont: [Font newFont:"Helvetica" size:12.0 style:0
    matrix:NX_FLIPPEDMATRIX]];
```

Notice that because a Text object draws in a flipped coordinate system, the matrix of the font it uses must match this orientation.

Since altering the font may change the number of characters that fit on a line, after receiving a **setFont:** message, the Text object recalculates the line layout and redraws the text.

When text is cut or copied to the pasteboard, it retains its font characteristics. When this text is pasted into another Text object, the pasted text can be displayed with its original font characteristics or it can assume those of the destination Text object. For many uses, it's more convenient to have the Text object use a single font. For example, a programmer writing a data-analysis application generally wouldn't want the added complexity of multiple font characteristics.

A **setMonoFont:**YES message makes a text object ignore the font information that accompanies pasted text. This is the default state of a Text object. The pasted text is displayed using the font characteristics of the first character of the existing text. The **isMonoFont** method reports whether a Text object will ignore the font information of pasted text.

Methods that change the font of selected text or read in text containing multiple fonts shouldn't be used with a Text object that is configured to contain only a single font. A text object configured in this way can display multiple fonts; however, each time text is cut or copied and then pasted, font information will be lost.

The Selection's Font

The Text class defines a set of methods that let you alter the selection's font:

setSelFont:
setSelFont:paraStyle:
setSelFontFamily:
setSelFontSize:
setSelFontStyle:

setSelFont: takes a Font object as an argument, allowing you to set several font characteristics simultaneously. The related method, **setSelFont:paraStyle:**, lets you additionally set the paragraph style of the selected text.

The next three methods each affect only one characteristic of the selection's font. **setSelFontFamily:** takes a single argument specifying a font name, such as Helvetica or Times-Roman. **setSelFontSize:** and **setSelFontStyle:** set the size and style of the selection's font.

As with **setFont:**, after a Text object resets the selection's font in response to any of the above messages, it recalculates the line breaks and redraws the text.

Drawing and the Text Class

A Text object, by default, is a transparent View: Sending it a **display** message doesn't cause it to paint every pixel within its frame rectangle. (View transparency is discussed under ªThe Display Methodsº in Chapter 7.) If a Text object's text is set in some way other than the keyboard (for example, by using a method such as **readText:** or **setText:**), the Text object draws only the characters on each line of text, not the pixels surrounding them. This allows you to draw text over another View without affecting more of that View than is necessary to display the characters.

However, when a Text object is receiving input from the keyboard, it always redraws the complete lineÐcharacters and backgroundÐthat the new characters fall on. If, after the user is through entering text, you want to restore the background, send the Text object a **displayFromOpaqueAncestor:::** message.

To force a Text object to repaint all the area within its frame rectangle whenever it receives a message to display itself, send it a **setOpaque:** message:

```
[myText setOpaque:YES];
```

A Text object, by default, draws black characters on a white background and marks the selection with by a light-gray highlight. You can change the gray value of the background, text, or selection by sending the appropriate messages.

The gray values of the background, text, and selection can be set to any value from 0.0 (represented by the constant NX_BLACK) to 1.0 (represented by the constant NX_WHITE) as the gray value argument of these methods. However, since only NX_WHITE, NX_LTGRAY, NX_DKGRAY, and NX_BLACK produce pure values on the NeXT computer's screen, these values result in the most legible text displays.

To invert the gray values of the standard text display, you would send these messages:

```
[myText setBackgroundGray:NX_BLACK];
[myText setTextGray:NX_WHITE];
[myText setSelGray:NX_DKGRAY];
```

The **backgroundGray**, **textGray**, and **selGray** methods return the current values for these three parameters.

The caret is always black, so setting the background gray value to black will obscure the caret.

Text Tables and Filter Functions

Much of a Text object's functionality is determined by the data it finds in certain tables and by the paradigms embodied in certain functions. The Text class provides methods that set these tables and functions, allowing you to change a Text object's behavior in fundamental ways without altering its class interface. This section introduces some of these tables and functions.

The Smart Cut and Paste Tables

As described in ªSmart Cut and Pasteº above, a Text object tries to maintain, during cut and paste operations, the proper relationship between words and their surrounding text. It does this by consulting the two tables that are referred to by its **preSelSmartTable** and the **postSelSmartTable** instance variables. The Text class provides the **setPreSelSmartTable:** and **preSelSmartTable** methods to set and return the former table and a

matching pair of methods to manage the latter table. These smart cut and paste tables define which characters a Text object should consider equivalent to a space.

For smart cut and paste to work, the selection must be the result of word selection. When you select a word or group of words and then cut the selection, the Text object removes the selection and the space to its left (if any). However, only the text is stored on the pasteboard; the space character is ignored. The Text object also records on the pasteboard that the cut text resulted from a smart cut operation.

When you paste the pasteboard's contents into the text, the Text object first checks to determine whether the pasteboard contains text that is the result of a smart cut or copy operation. If it does, the Text object looks at the characters that border the right and left sides of the insertion point or selection to be replaced. If the character on the left is in its smart left paste table, it pastes the word in without adding or subtracting anything on that side. If the character isn't in the left table, the Text object adds a space on that side when it pastes the word. It uses the same process on the right.

Here are the tables that the Text object uses by default:

```
unsigned char  NXSmartLeft[] = {' ', NX_EMSPACE, NX_ENSPACE,
    NX_THINSPACE, NX_FIGSPACE, '\n', '(', '\t', '[', '\320 ', '\"',
    '\'', '\0'};

unsigned char  NXSmartRight[] = {' ', NX_EMSPACE, NX_ENSPACE,
    NX_THINSPACE, NX_FIGSPACE, '\n', ')', '\t', ']', '.', ',', ';',
    ':', '?', '\'', '!', '\"', '\0'};
```

So, for example, pasting a word immediately to the right of a Tab character won't cause a Text object to add a space character on that side. Similarly, pasting a word immediately to the left of a colon doesn't cause a Text object to add a space between the word and the colon.

If you are going to use a Text object as an editor for C language programs, for example, you might prefer both tables to look like this:

```
unsigned char  smartForC[] = {' ', '\200', '\201', '\202', '\203',
    '\n', '\t', '!', '\"', '#', '$', '%', '&', ''', '(', ')', '*',
    '+', ',', '-', '.', '/', ':', ';', '<', '=', '>', '?', '@', '[',
    '\', ']', '^', '_', '`', '{', '|', '}', '~', '\320'}
```

To set this table for both the preSelSmartTable and the postSelSmartTable, you'd send these messages:

```
[text setPreSelSmartTable:smartForC];
[text setPostSelSmartTable:smartForC];
```

This table illustrates the different effects of the default tables in comparison to the C language table:

		Before Cut		After Cut	After Paste
Default	!word!	!!	! word!		
C Table	!word!	!!	!word!		
Default	(word(	((	(word (		
C Table	(word(	((	(word(		
Default	*word*	**	* word *		
C Table	*word*	**	*word*		

The Click Table

A *click table* is a table a Text object consults to determine word boundaries for word selection. It's used whenever the user double-clicks a word, Shift-clicks to extend a word selection, or double-clicks and then drags to select a series of words. The default click table is designed to be used with standard English text. The **setClickTable:** and **clickTable** methods give you control of your Text object's click table.

The Break Table

A *break table* is a table that a Text object uses to determine word boundaries for line breaks. It's similar to the click table, but has a different view of word boundaries. For example, if you insert enough text in the middle of a sentence to push the last word of the sentence onto the next line, the standard break table ensures that both the word and the final punctuation wrap to the next line. In contrast, if you double-click the last word of the sentence, the standard click table ensures that only the word, and not the final punctuation, is selected.

The **setBreakTable:** and **breakTable** methods give you access to the break table, although most programmers will rarely need a break table other than the default one.

Filter Functions

A text object supports two types of input filters: a character filter and a text filter. Character filters remap one character code to another when necessary. Text filters allow for more extensive manipulation of the character or characters to be added to the text.

When the user enters a character from the keyboard, a Text object calls a character filter function to examine the entry. It can also optionally call a text filter function if one is installed. Neither type of filter examines characters that are read into the text from a file or from the pasteboard.

Depending on the current character filter function and the value of the entered character, the Text object adds the character to the list of those to be displayed, interprets the character as a command, or ignores the character.

Unlike a character filter, which only receives the individual character that's entered, a text filter additionally receives information about the state of the Text object. Based on this information, the text filter can change a number of variables including the content of the entered or existing text and the location of insertion point.

Character Filter Functions

The Text class provides two character filters, **NXFieldFilter()** and **NXEditorFilter()**, and you can write your own if necessary. **NXFieldFilter()** can be used to implement electronic forms. When the user presses Return, Tab, or Shift-Tab, **NXFieldFilter()** causes the Text object to cease being the first responder, letting some other object assume that role. **NXEditorFilter()** is designed to accept a wider variety of characters. It lets the user enter Tab and Return characters directly into the text.

By default, the Text object uses **NXFieldFilter()** as its character filter. You can change its filter by sending a **setCharFilter** message:

```
[myText setCharFilter:NXEditorFilter];
```

Character filter functions operate by reassigning the codes of certain characters before sending the code on to the Text object. **NXFieldFilter()** and **NXEditorFilter()** reassign most codes less than 0x20 (the ASCII space character) to 0x00. Since a Text object ignores input having a value of 0x00, most control characters are excluded from the text.

Codes generated by Return, Tab, and Shift-Tab (back tab) are given special treatment. Depending on the filter, these codes can be sent unaltered to the Text Object or they can be first remapped to the corresponding constant, as defined in **Text.h**:

NX_RETURN
NX_TAB
NX_BACKTAB

When a Text object receives a character code having one of these defined values, it attempts to end its status as first responder. If it has a delegate, the Text object sends the delegate a **textWillEnd:** message, and assuming the delegate allows the change, the Text object then sends a **textDidEnd:endChar:** message. This latter message includes a value that the delegate can check against these constants to determine which key caused the Text object to lose first-responder status:

```
- textDidEnd:sender endChar:(unsigned short)whyEnd
{
    switch (whyEnd) {
    case NX_RETURN:
    /* send a message based on contents of the Text object */
        break;
    case NX_TAB:
    /* make the next text object the first responder */
        break;
    case NX_BACKTAB:
    /* make the previous text object the first responder */
        break;
    }
}
```

The Matrix class uses code similar to this to let the user tab among the various fields.

NXFieldFilter() is the default character filter function for the Text class. The listing below details how **NXFieldFilter()** remaps the codes generated by Delete, Return, and Tab.

```
NXFieldFilter(unsigned short theChar, int flags)
{
    if (flags & NX_COMMANDMASK)
```

```
    {
        theChar = 0;
    } else {
        if (theChar == NX_DELETE)
            theChar = NX_BACKSPACE;
        else if (theChar == NX_CR) {
            theChar = (flags & NX_SHIFTMASK) ? '\n' : NX_RETURN;
        } else if (theChar == '\t') {
            theChar = (flags & NX_SHIFTMASK) ? NX_BACKTAB : NX_TAB;
        } else if ((theChar < ' ') && (theChar != '\n') &&
                    (theChar != NX_BACKSPACE))
            theChar = 0;
    }
    return (theChar);
}
```

Notice that the codes generated by Return and Tab aren't remapped if Shift is down when the key is pressed. This allows the user to enter these characters even if **NXFieldFilter()** is in use.

The Text class also provides **NXEditorFilter()**, a character filter function for more general text editing:

```
NXEditorFilter(unsigned short theChar, int flags)
{
    if (flags & NX_COMMANDMASK)
    {
        theChar = 0;
    } else {
        if (theChar == NX_DELETE)
            theChar = NX_BACKSPACE;
        else if (theChar == NX_CR) {
            theChar = '\n';
        } else if (theChar == '\t') {
            ;
        } else if ((theChar < ' ') && (theChar != '\n') &&
                    (theChar != NX_BACKSPACE))
            theChar = 0;
    }
    return (theChar);
}
```

Notice that **NXEditorFilter()** passes the codes generated by Return and Tab through to the Text object. Shift has no effect on how these codes are handled.

You might want to write your own character filter function. For example, **Text.h** defines four other constants that cause a Text object to attempt to resign first responder status:

NX_LEFT
NX_RIGHT
NX_UP
NX_DOWN

By writing a character filter function that remaps certain Control sequences to these constants (such as Control-H, or 0x08, to NX_UP), you could provide alternate movement commands. Alternatively, with the help of the Text object's delegate, the new character filter could call a help system whenever the user entered such a Control sequence.

Text Filter Functions

The Text class provides for a text filter function although, by default, a text object doesn't use one. To install a text filter function, send a **setTextFilter:** message:

```
[myText setTextFilter:FilterText];
```

The **textFilter** method returns the current text filter.

Once a text filter function is installed, it's called whenever text is entered from the keyboard. The filter receives four arguments: the **id** of the caller, a pointer to the character being entered, a pointer to the length of the text to be inserted, and the character position of the insertion. The filter can alter the inserted text, or the Text object's preexisting text, in any way. Whatever the alteration, the text filter must inform the Text object of new text to be inserted and the length of the inserted matter.

For example, the following simple text filter examines incoming characters for the occurrence of the ª&º character. If it finds one, it then checks the character immediately preceding the current character position. If the preceding character is a space, it replaces the ª&º with the word ªandº. Note that if it makes such a

substitution, it returns the length of the substitute string in the address referred to by **inputLength**:

```
char  sample[3];
char  model[] =" &";
char  substitute[] ="and";

char *FilterText(id textObj, char *inputText, int *inputLength,
                    int position)
{
    if (position > 0) {
        /* get character that precedes this one */
        [textObj getSubstring:sample start:position-1 length:1];

        /* add this character and null terminator */
        sample[1] = *inputText;
        sample[2] = '';

        /* compare sample with model */
        if (!strcmp(sample, model)) {
            /* make substitution if they are equal */
            *inputLength = 3;
            return(substitute);
        }
    }
    /* otherwise, simply return original character */
    return (inputText);
}
```

A Text Object in a Scrolling View

To accommodate large blocks of text, a Text object is commonly made a subview of a ScrollView. In this way, the user can scroll into view portions of the text that lie beyond the ScrollView's frame. A Text object must, however, be properly configured before it will cooperate with the ScrollView.

This example program creates a ScrollView within a Window and installs a Text object as the ScrollView's

document view:

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

main(int argc, char *argv)
{
    id theWindow, theScrollView, myText;
    NXRect aRect, contentRect;
    NXSize aSize;

    NXApp = [Application new];    /* create Application object */

    NXSetRect(&aRect, 100.0, 350.0, 300.0, 300.0);
    theWindow = [Window newContent:&aRect         /* create Window */
                    style:NX_TITLEDSTYLE          /* object */
                    backing:NX_BUFFERED
                    buttonMask:NX_ALLBUTTONS
                    defer:NO];
    [theWindow setBackgroundGray:NX_WHITE];

    theScrollView = [ScrollView newFrame:&aRect]; /* create */
    [theScrollView setVertScrollerRequired:YES];  /* a */
    [theScrollView setHorizScrollerRequired:NO];  /*ScrollView*/

    [theWindow setContentView:theScrollView];

    contentRect = aRect;       /* find size of content view */
    [ScrollView getContentSize:&(contentRect.size)
            forFrameSize:&(aRect.size)
            horizScroller:NO
            vertScroller:YES
            borderType:NX_NOBORDER];

    myText = [Text newFrame:&contentRect         /* create a Text */
                    text:NULL                    /* object */
                    alignment:NX_LEFTALIGNED];
    [myText notifyAncestorWhenFrameChanged:YES]; /*configure*/
    [myText setVertResizable:YES];               /* it to work with */
    [myText setHorizResizable:NO];               /* the ScrollView */
```

```
    aSize.width = 0.0;      /* let the Text object get no smaller */
    aSize.height = contentRect.size.height;     /* than height of */
    [myText setMinSize:&aSize];      /* ScrollView's content view */

    aSize.width = contentRect.size.width;   /* let the Text */
    aSize.height = 1000000.                 /* object's height grow */
    [myText setMaxSize:&aSize];             /* to large value */

    [theScrollView setDocView:myText];      /* make myText the doc.*/
    [[myText superview] setAutoresizeSubviews:YES];   /* view */
    [[myText superview] setAutosizing:NX_HEIGHTSIZABLE |
        NX_WIDTHSIZABLE];   /* notify Text object if Window resized */

    [theWindow display];                  /* display the window in */
    [theWindow orderFront:nil];           /* front of other windows; */
    [theWindow makeKeyWindow];            /* make it the key window */

    [NXApp run];                          /* start the Application */
}
```

In this example, a Text object is made the document view of a ScrollView. The ScrollView only allows scrolling in the vertical direction, so the Text object is also constrained to grow only vertically. The Text object's frame is sized to fit within the scrolling portion of the ScrollView. When a user adds enough text to the Text object to cause its frame to grow, the **notifyAncestorWhenFrameChanged:** message ensures that the ScrollView is notified of the change. The ScrollView automatically scrolls the new line of text into view and then displays a knob to allow the user to access other portions of the document view. The knob will disappear if the user deletes enough text to shrink the Text object to its original, and minimum, size.

Reusing a Text Object

As an efficiency, a single Text object can be reused: It can be made to draw in various Views or various locations within one View. In fact, you can make use of a Window's field editor, the Text object that the Application Kit uses to draw text within standard Kit objects. Using one Text object rather than several saves

both startup time and memory.

Archiving a Text Object

The **write:** method writes the Text object's instance variables out to the archive stream. It doesn't, however, maintain the values of the following instance variables if you've reassigned their values to functions or tables not provided by the Text class:

charFilterFunc
scanFunc
drawFunc
charCategoryTable
preSelSmartTable
postSelSmartTable
breakTable
clickTable

The **read:** method reads the Text object's instance variables into memory from the archive stream. Once the Text object is reestablished in memory, you may want to reset some of the values of the instance variables above.

One way to do this is to send the appropriate messages to the Text object from another object's awake method. **awake** messages are only sent after the Text object and all the objects it refers to have been read in, ensuring that the Text object is ready to receive the initializing messages. In the same way, you can send a **setSel::** message to reestablish the selection.

Rich Text Format Support

A Text object can encode and decode a subset of the formatting commands defined in RTF; the following table

summarizes that support. For example, a Text object can read and properly display the characters associated with a **\b** control word, making those characters have a bold attribute. It can also embed this control word along with the affected characters when it writes the file to the disk. For other control words, such as **\margl***n,* a Text object will display the associated characters properly but won't write the control word when it writes the characters to a file. When reading text from a file, a Text object ignores any RTF control word that's not listed in the following table.

Control Word	Read/Write
\ansi	yes/yes
\paperw*n*	yes/yes
\margl*n*	yes/yes
\margr*n*	yes/yes
\pard	yes/no
\s*n*	yes/no
\ql	yes/yes
\qr	yes/yes
\qc	yes/yes
\fi*n*	yes/yes
\li*n*	yes/yes
\b	yes/yes
\i	yes/yes
\f*n*	yes/yes
\fs*n*	yes/yes
\up*n*	yes/yes
\dn*n*	yes/yes
\par	yes/yes
\tab	yes/yes

The Text class doesn't write the **\pard** control word; instead, it manipulates groupings to restore paragraph default values when needed. In addition, the Text class interprets the **\sect**, **\page**, and **\line** control words as Return characters. The Text class recognizes all installed fonts. See *Rich Text Format Specification* by Microsoft Corporation for more information about RTF.

The Box Class

A Box is a View that visually groups other Views. As shown in Figure 9-3, a typical Box displays a border and a title. Box objects are often used to group choices that a user can make in a panel. For example, in a Find panel, one Box may surround buttons that set the scope of the search; another may enclose the switches that specify whether the case of letters should be ignored and whether the text to be found should be treated as a regular expression.

F5.eps ,

Figure 9-3. A Typical Box

A Box's border encloses its contents. After changing the Box's contents, you can resize the Box to fit. You can also change how the Box looks by modifying its border type and the title's position and font. The following two sections discuss how to accomplish these tasks. However, you may want to use Interface Builder to create and set the initial characteristics of a Box, since that's easy to do. Your application code could then use the methods discussed in the following sections to modify the Box dynamically at appropriate moments in your program.

Creating and Modifying a Box Object

The **newFrame:** class method returns a new instance of the Box class with the location, height, and width specified by its argument:

```
NXRect myRect;

NXSetRect (&myRect, 200.0, 200.0, 500.0, 500.0);
```

```
myBox = [Box newFrame:&myRect];
```

Since you'll probably be adding to the Box's initially empty contents and then resizing the Box, its original size isn't critical. It's easy to make the Box larger or smaller as needed to encompass its contents. The ªResizing the Boxº section below explains how to do this.

A Box object defines a set of rectangles that determine how component parts are laid out. These rectangles are shown in Figure 9-4.

F4.eps ,

Figure 9-4. The Layout Rectangles of a Box

The external boundary of the Box is defined by its frame rectangle. Within this boundary lies the border rectangle; its location is determined by the location and size of the title rectangle, the rectangle that encloses the title. If you choose to display the title above the top edge of the border, for example, you will cause the border rectangle to shrink. The innermost, or content, rectangle encompasses the contents of the Box. You can set the value of the offsets, which determine the amount of horizontal and vertical space between the content rectangle and the border rectangle.

The Border

A Box's border can be marked with a line, a bezel, a groove, or nothing at all. Figure 9-5 shows these border types and the corresponding constants you can use to set the type.

F3.eps ,

Figure 9-5. Border Types

By default, a Box has a grooved border. To change this, you send a **setBorderType:** message to the Box, with one of the four constants as its argument. For example, to set the border type to a bezel:

```
[myBox setBorderType:NX_BEZEL];
```

You can also query the Box about its border type, with the **borderType** method:

```
int  myBorderType;
myBorderType = [myBox borderType];
```

Since the border is drawn inside the Box, it will slightly reduce the space available for the content rectangle. A borderless Box doesn't affect the content rectangle, but lines, bezels, and grooves reduce both the width and the height of the content rectangle by 2.0, 4.0, and 6.0, respectively. In addition to setting the border type, the **setBorderType:** method recalculates the size of the content rectangle based on the width of the new border.

The Title

You can have a Box with no title or one with a title positioned above, below, or intersected by the border at either the top or the bottom of the Box. Figure 9-6 shows the possible title positions and the corresponding constants you can use to set the position. The default position is NX_ATTOP.

F2.eps ,

Figure 9-6. Title Positions

You set the title position by specifying one of the seven constants with the **setTitlePosition:** method. The following example sets the position to be above the top of the border:

```
[myBox setTitlePosition:NX_ABOVETOP];
```

The **setTitlePosition:** method adjusts the rectangles of the Box so that they're positioned correctly relative to the title.

A Box's default title is ªTitleº. To change this, you send the Box a **setTitle:** message and specify the text you

want displayed as the title:

```
const char  *myTitle ="Find Options";
[myBox setTitle:myTitle];
```

In addition to changing the text, the **setTitle:** method recalculates the size of the title rectangle. If the title is longer than the width of the Box, the title rectangle's width is reduced to the size of the Box and the text is clipped. **setTitle:** also provides a small cushion of space on the top and bottom of the title so that it doesn't actually touch the Box's border.

By default, the title will be displayed using 12-point Helvetica. However, you can use the **setFont:** method to specify a new font, as shown below. (You could also use the defaults mechanism to change the system font, but this may affect more than the title of your Box; see ªThe Defaults Systemº in Chapter 10, ªSupport Objects and Functions.º)

```
[myBox setFont:aNewFontObj];
```

The **setFont:** method recalculates the Box's rectangles. If you increased the font size significantly, for example, the title rectangle would increase; the content rectangle and possibly the border rectangle (if the title is outside of or intersected by the border) would decrease to accommodate the larger title rectangle.

You can also query a Box about its title, title position, and font:

```
id     myFontObject;
char  *myTitle;
int    myTitlePosition;

myTitle = [myBox title];
myTitlePosition = [myBox titlePosition];
myFontObject = [myBox font];
```

The **title** method returns a pointer to the Box's title. **titlePosition** returns one of the seven constants that correspond to the title's position, and **font** returns the **id** of the font object used to display the title.

The Offsets

The content rectangle of a Box can be horizontally and vertically offset inside the border. This allows you to

adjust the space between the contents and the border and to make the horizontal offset different than the vertical offset. As shown above in Figure 9-4, the two vertical offsets (at the top and bottom of the Box) are equal, and the horizontal offsets at each side are equal. When you set the offsets, the size of the Box rather than the content rectangle is changed to accommodate the offsets.

Initially, a Box is created with default offsets of 5.0. To set new offsets, you supply them as arguments to the **setOffsets:** method, as shown below. (Although you can specify negative offset values, the contents of the Box or the Box itself may be clipped.)

```
[myBox setOffsets:40.0 :0.0];
```

The first argument for the **setOffsets** method refers to the horizontal offset, and the second refers to the vertical offset. In this example, if the previous offset values were the default 5.0, the horizontal dimension of the Box's frame rectangle will increase by a total of 70.0 and the vertical dimension will decrease by 10.0. After changing the offsets, you'll want to resize the box using the **setFrameFromContentFrame** method, which is described in more detail in the ªResizing the Boxº section below.

You can obtain the current values of a Box's offsets by querying the Box:

```
NXSize  mySize;
[myBox getOffsets:&mySize];
```

The offsets will be placed in **mySize**, which must be an NXSize structure.

The Contents of the Box

As a graphic object, a Box's purpose is to visually group its contents, typically by displaying a border and a categorizing title. As discussed above, you can modify the border and title in several ways. You can also add to the Box's contents and then resize it so that it comfortably encloses them. For example, you may want to use a Matrix of switches that changes the number of Cells it displays. The next two sections discuss how to manage a Box's contents and how to resize the Box.

Box's View Hierarchy

Views are added to the Box's contents by making them subviews. Here **newSubview** is added to **myBox**:

```
[myBox addSubview:newSubview];
```

This actually makes **newSubview** a subview of the content view, which corresponds to the Box's content rectangle. The content view groups the Box's contents and makes sure that they're displayed only within the content rectangle. Since the content view is a subview of Box, the frame rectangles of Views added to the Box should reflect their position in the content rectangle, not the Box's frame rectangle. After you've added a subview, you'll probably want to use the **sizeToFit** method (described below) to adjust the Box's size to accommodate its new subview.

To replace the existing content view, you send a **setContentView:** message to the Box, giving it the new content view as the argument:

```
[myBox setContentView:newContentView];
```

The old content view is returned so that you can either free it or use it in another view hierarchy. Since **setContentView:** recalculates the size of the Box based on the size of the new content view, you don't need to resize it.

Resizing the Box

If you've added a subview to the Box's contents, you should use the **sizeToFit** method to adjust the size of the Box. **sizeToFit** calculates the appropriate size for the content rectangle so that it just encloses all the content view's subviews. It then uses the **setFrameFromContentFrame:** method to resize the Box to match the new content rectangle.

You can also send the **setFrameFromContentFrame:** method directly to the Box to specify a new location and size for the content rectangle:

```
NXRect myContentRectangle;

NXSetRect (&myContentRectangle, 100.0, 100.0, 300.0, 300.0);
[myBox setFrameFromContentFrame:&myContentRectangle];
```

The Box is resized and relocated so that its content view has the same dimensions as the specified content

rectangle, which is in the coordinate system of the Box's superview.

You can also resize the Box from the outside in, using the **sizeTo::** method. The two arguments specify the new width and height of the Box. The Box's layout rectangles are then recalculated to fit inside this new boundary. If the new size of the Box is too small for its content view (taking into account the offsets), the drawing that the content view displays will be clipped at the Box's boundaries.