CHAPTER 16

Text Manipulation with Lingo

In This Chapter

Setting font, font color, font style, and font size

Working with chunk expressions

Using hyperlinks

Formatting paragraphs

Selecting text

HTML and RTF of text members

Field-only properties

Even in the world of multimedia, where pictures and sounds reign supreme, creating and manipulating strings of text is a large part of programming with Director. With a little bit of Lingo, you can control the properties of text in the Cast window or on your Stage. In this chapter, you learn to control every attribute of text through Lingo that you can set through the user interface, plus a few extras.

Understanding the Properties of Text

Director 8 has two ways of implementing text in the user interface — through field cast members and text cast members. For brevity, we refer to text cast members as *text members* and field cast members as *fields*. Text in a variable or written in the code (between quotes) is referred to as a *string*. Support of fields is mainly for historical reasons, but they still have their advantages (see the "Text Cast Members versus Field Cast Members" sidebar).

Setting the foreground color

The foreground color is the color of the text in the field. The default is black text on a white background.

Text Cast Members versus Field Cast Members

There are important differences and confusing similarities between fields and text members. We need to briefly review software history to shed some light on them for both old and new users.

In Director 4 and earlier, fields were the only way to get text on the screen (other than using bitmaps, which aren't really text but images). The type of member was `#text`.

In Director 5 and 6, if you wanted text that was editable at run-time by Lingo or by the user, you needed to use fields. If you wanted antialiased text, you needed to use text members. The problem with text members was that during playback in Shockwave or a projector, you could not edit them, because they were converted to bitmaps. The type of this new member was `#richText`. The type of a field now returned was `#field` instead of `#text`.

In Director 7, you could edit text members at run-time for the first time. You could also change their attributes, such as fontStyle, font, and alignment. Text members can now do everything fields can do (with minor exceptions) and more. If you need your text to rotate, skew, flip, or antialias, or if you need to have hyperlinks, change `charSpacing` or paragraph spacing, or get its picture, or have its quads manipulated, use text members. If you want to save around 500 bytes per cast member, then use fields. With all that you can do with text, we can't think of a reasonable argument to use fields, except in situations in which you are dealing with pre-Director 7 movies or maybe storing information within a movie to be saved. Since Director 7, a test of the text member's type now returns `#text` instead of `#richText`; fields still remain `#field`.

Getting and setting the foreColor

You can change the foreground color of text in a member using the `foreColor of member` property. The syntax is:

```
set the foreColor of member targetMember to colorNumber
```

In this statement, `targetMember` is replaced with either the cast member name or number. The `colorNumber` is replaced with an index number representing a color from the current color palette. The value should be set between 0 and 255—a higher number wraps back around; for example, 256 displays color 0, 257 displays color 1, and so on.

On the CD-ROM

The text1.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Open a new movie and create a text cast member on the Stage, or open text1.dir on the CD-ROM. In the Message window, type the following:

```
set the foreColor of member 1 to 35
```

This changes the text to red (35 happens to be red on both Macintosh and Windows platforms). You can also use dot notation, which has this syntax:

```
member(targetMember) = colorNumber
```

To use it in our current movie, type the following in the Message window:

```
member(1).foreColor = 255
```

It now changes to black.

Getting and setting the color

Another way to set the foreground color of text is to use the new color data type (first discussed in Chapter 12). You can create `rgb` or `paletteIndex` color objects and assign the color to the text. In the Message window, type the following:

```
myColor = color(#paletteIndex, 35)
put myColor
-- paletteIndex( 35 )
member(1).color = myColor
```

Make sure that you use the parentheses around the member number! Now, try testing the color of the member, as follows:

```
put the color of member 1
-- rgb( 255, 0, 0 )
```

Even though you assigned a `paletteIndex` color, the color property always returns a color in the form `rgb(`*red*`,` *green*`,` *blue*`)`. But the index can still be retrieved with the `foreColor` property:

```
put the foreColor of member 1
-- 35
```

Caution

Getting the `foreColor` of the member, as just shown, will return the foreColor of the first character in the text member.

Oddly, if you try to get the `paletteIndex` of this member, it will return 255, not 35, even if all of the text is set to red.

```
put the paletteIndex of member 1
-- paletteIndex( 255 )
```

Setting the background color

The background color is the color behind the type — the default is black type on a white background.

Getting and setting the backColor

It probably comes as no surprise that you can set the background color of the text as well as the foreground color. The syntax for setting the background color is:

```
set the backColor of member targetMember to colorNumber
```

To use it with the preceding example, change the type to white and the background to red by typing the following code:

```
set the foreColor of member 1 to 0
set the backColor of member 1 to 35
```

Using dot notation, change the `backColor` to black, as follows:

```
member(1).backColor = 255
```

Getting and setting the bgColor

If you want to use the color data type, then you need to set the `bgColor` instead of the `backColor`:

```
set the bgColor of member(1) to myColor
member(1).bgColor = myColor
```

The background changes to red, assuming `myColor` still holds `paletteIndex(35)`, as you specified earlier. You can set rgb colors as well.

Setting the fontSize

To change the type size of the text stored in a field, use the `fontSize of` member property. The syntax for both standard and dot notation is the following:

```
set the fontSize of member targetMember to size
member(targetMember).fontSize = size
```

In this statement, `targetMember` is replaced with either the cast member name or number. The size parameter is replaced with the desired font size in points.

On the CD-ROM

The text1.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Open a new movie and create a text cast member on the Stage, or open text1.dir on the CD-ROM. In the Message window, type the following:

```
member(1).fontSize = 24
```

The type changes to the smaller point size. Now, change it again, as follows:

```
set the fontSize of member 1 to 36
```

As with the color properties, you can get this value as well as set it:

```
put the fontSize of member 1
-- 36
put member(1).fontSize
-- 36
```

Setting the font

You can change the typeface of text by using the `font of member` property. The syntax for both standard and dot notation is the following:

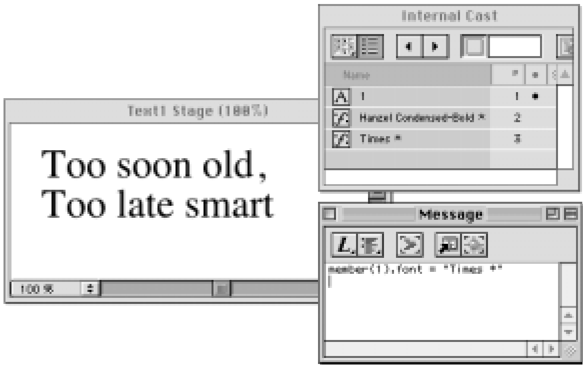
```
set the font of member targetMember to fontName
member(targetMember ).font = fontName
```

In this statement, `targetMember` is replaced with either the cast member name or number. The `fontName` parameter is replaced with the name of a font.

Open text1.dir on the CD-ROM. In the Message window, type the following:

```
set the font of member 1 to "Times *"
```

This changes the typeface to the embedded font “Times *”, which is cast member 3, as shown in Figure 16-1.



Name	#
1	1
Hanzel Condensed-Bold *	2
Times *	3

```
member(1).font = "Times *"
```

Figure 16-1: Changing the font of member 1 to the embedded font “Times *”.

To change the typeface back, use dot notation:

```
member(1).font = "Hanzel Condensed-Bold *"
```

Knowing that “Hanzel Condensed-Bold *” is in member 2, you could write the following code:

```
member(1).font = the name of member 2
```

Tip

Referencing members by their position in the cast is not a good idea in your code; it is safer to reference them by name. (Otherwise, every time you move the member, you have to change your code!) When you are working in the Message window, experimenting can be fun, but sometimes it’s nice to do things the easy way.

Missing fonts

The `missingFonts` text cast member property returns a list of font names that are referenced by a member but are not in the system.

Set the font of member 1 to one that is not in your system:

```
set the font of member 1 to "abc"
```

The text member remembers the font it should be, but because it is not there, it is displayed in Helvetica. If you `put` the `missingFonts`, you see it is there:

```
put the missingFonts of member 1
-- ["abc"]
```

Substituting fonts

The `substituteFont` text member command substitutes one font in a member for another. The syntax for both standard and dot notation is as follows:

```
substituteFont(member targetMember, fontName1, fontName2)
member(targetMember).substituteFont(fontName1, fontName2)
```

If you want to deal with the missing font from the earlier example, you should write the following:

```
substituteFont(member 1, "abc", "Hanzel Condensed-Bold *")
```

Adding Shockfonts

Shockfonts are fonts embedded into your movie. With the `recordFont` command, you can add fonts through Lingo. The syntax is the following:

```
recordFont(fontMember, font, [style], [bitmapSizes],
characterSubset)
```

The various parameters for the `recordFont` command are:

- ✦ `fontMember` is a font member.
- ✦ `font` is a string, the name of the font you want to record (embed).

✦ `[style]` is a list of symbols. They are the style of the font you want to include, such as `#plain`, `#bold`, and `#italic`. This is an optional parameter; if left off or set to an empty list, only `#plain` is used.

✦ `[bitmapSizes]` is a list of integers. They are the point sizes of bitmapped fonts that you want to include. For smaller text sizes, bitmapped fonts are supposed to be more readable. This is an optional parameter; if left off or set to an empty list, it does not include any bitmaps. Including bitmaps increases the file size.

✦ `characterSubset` is a string. It is the list of characters to include in the font. If you need only a few characters, this can save you space. This is an optional parameter; if left off, it includes all of the characters in the font.

To use `recordFont`, you must use a font member. If you don't have one, you can create one by using the `new` function, as shown here:

```
myFont = new(#font)
recordFont(myFont, "Times", [#bold], [18, 24], "Mars Landing")
```

If you don't want to specify any optional parameters, you can write this code:

```
recordFont(myFont, "Times")
```

If you just want to use the last optional parameter, you need to remember to put empty lists in for the other optional parameters (because they take lists), as follows:

```
recordFont(myFont, "Times", [], [], "Mars Landing")
```

Determining fonts available

There are two undocumented commands for determining the fonts in your system. The first is the `fontList` function. The syntax for both standard and dot notation is:

```
fontList(fontMember)
fontMember.fontList()
```

You can use any font member to get this information. In the movie text1.dir, type the following command in the Message window and see these results:

```
put fontList(member "Times *").count
-- ["Abadi MT Condensed Light", "Arial", "Arial Black",
"ASI_Mono", "ASI_System", "Book Antiqua", "Calisto MT",
"Century Gothic", "Comic Sans MS", "Copperplate Gothic Bold",
"Copperplate Gothic Light", "Courier", "Courier New",
"Fixedsys", "Hanzel Condensed-Bold *", "Impact", "Lucida
Console", "Lucida Handwriting", "Lucida Sans", "Lucida Sans
Unicode", "Marlett", "Matisse ITC", "MS Sans Serif", "MS
Serif", "News Gothic MT", "OCR A Extended", "Small Fonts",
"Symbol", "System", "Tahoma", "Tempus Sans ITC", "Terminal",
"Times *", "Times New Roman", "Verdana", "Webdings",
"Westminster", "Wingdings"]
```

The list returned is a list of all fonts installed.

The next function is called the `outLineFontList`. This function returns all fonts in the system that can be used to create Shockfonts using `recordFont`. The syntax for both standard and dot notation is:

```
outLineFontList(fontMember)
fontMember.outLineFontList()
```

Just as you can use any font member as an argument for the `fontList` function, the `outLineFontList` also takes any font member. In the movie text1.dir, you type the following `put` command in the Message window and see results similar to those shown here:

```
put outLineFontList(member "Times *").count
-- ["Abadi MT Condensed Light", "Arial", "Arial Black", "Book
Antiqua", "Calisto MT", "Century Gothic", "Comic Sans MS",
"Copperplate Gothic Bold", "Copperplate Gothic Light", "Courier
New", "Hanzel Condensed-Bold *", "Impact", "Lucida Console",
"Lucida Handwriting", "Lucida Sans", "Lucida Sans Unicode",
"Marlett", "Matisse ITC", "News Gothic MT", "OCR A Extended",
"Symbol", "Tahoma", "Tempus Sans ITC", "Times *", "Times New
Roman", "Verdana", "Webdings", "Westminster", "Wingdings"]
```

Your results will differ from these, because they are based on the fonts in your system. Note that the `fontList` in this example has 38 items and the `outLineFont List` has only 29.

Note

The preceding results for `fontList` and `outLineFontList` were done on a system running Windows 98.

Caution

The `fontList` and the `outLineFontList` are not documented functions. Use them at your own risk.

Setting the style

To change the style of text, use the `fontStyle of` member property. The syntax for both standard and dot notation is:

```
set the fontStyle of member targetMember to styleName
member(targetMember ) = styleName
```

In this statement, `targetMember` is replaced with either the cast member name or number. The `styleName` parameter is replaced with a list of styles.

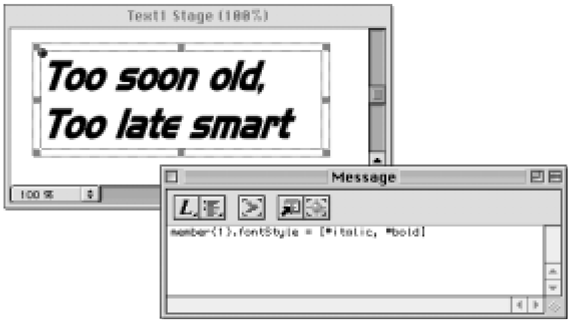
On the CD-ROM

The text1.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Open the text1.dir movie on the CD-ROM. In the Message window, type the following:

```
set the fontStyle of member 1 to [#italic, #bold]
```

The styles are in the form of symbols, and you can specify more than one style in the list. In the preceding example, the text is changed at once to be bold and italic, as shown in Figure 16-2.



Too soon old,
Too late smart

```
member(1).fontStyle = [#italic, #bold]
```

Figure 16-2: The fontStyle can be changed to italic and bold in one step.

There are four styles supported: `#plain`, `#bold`, `#italic`, and `#underline`. Setting a member to `[#plain]` removes all styles. But setting them one by one has an additive effect. Set the text to plain, and then, one statement at a time, set it to be bold, italic, and underline, and use dot notation, as shown here:

```
member(1).fontStyle = [#plain] -- gets rid of all styles
member(1).fontStyle = [#bold]  -- now it is bold
member(1).fontStyle = [#italic] -- now it is bold and italic
member(1).fontStyle = [#underline] -- now it is bold, italic
and underlined
```

Using the autoTab property

The autoTab property determines whether the Tab key makes the next editable text active. This property works with both fields and text members. It can be set to TRUE or FALSE. The order in which the tabbing occurs is based on the sprite channel numbers, not position on the Stage. Following are examples of dot and standard notation for turning the autoTab property OFF and then ON again for member “myText”:

```
member("myText").autoTab = FALSE
set the autoTab of member "myText" to TRUE
```

Using the picture property

You can create bitmapped images of text members with the picture property. This property works with both fields and text members. To create a picture and place it into a new bitmap member, type the following:

```
(new(#bitmap)).picture = member("myText").picture
```

The bit depth of the bitmap created is that of the monitor. If you want to capture the reference of the member created with the new(#bitmap) function, you can do the preceding in several steps. The advantage of having the member reference in a variable is that it enables you to know where that bitmap was put, and to do things such as name the new member:

```
set bm to new(#bitmap)
bm.picture = member(1).picture -- member(1) is a text member
bm.name = member(1).name && "bitmap" -- assumes member(1) has a
name
```

Using the boxType property

The boxType is the type of text box used for the field or text member. Options are #adjust, #scroll, #fixed, and #limit (#limit is for fields only).

Using the editable property

This property enables you to change whether a field or text member is editable.

Using Chunk Expressions

So far, you have used the properties of text to change the entire contents of a text member. By using Director’s chunk expressions, you can take a “bite” out of a text member and manipulate that chunk without changing the entire contents of the member. In word processing, you can change or modify a specific character, word, line, or paragraph of text. In Director, you can also control a chunk of text, such as a specific character, word, item, or line.

Understanding strings

Your computer looks at text as a string of characters. For instance, the following string of characters makes up a quotation:

"They are ill discoverers that think there is no land, when they can see nothing but sea." — Sir Francis Bacon (from *Advancement of Learning*).

This sample string includes not only alphabetical characters (A, a, B, b, C, c, and so on), but also punctuation (quotation marks, period, and comma) and special characters (left and right parentheses), plus spaces that separate words. When you work with chunk expressions, don't forget that spaces, punctuation, and special characters are all counted as part of a string.

Guidelines for working with strings

This section provides some helpful rules to keep in mind when you're working with strings in Director.

To work effectively with strings, you must remember that:

- ✦ Strings are always surrounded by quotation marks. The quotation marks used in the Lingo language to denote a string are straight quotation marks, like those used for inches, not the "curly" quotes used in typesetting.
- ✦ You can insert a string into fields, text members, or into a variable by using the `put` command.

On the CD-ROM

The text2.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Open text2.dir on the CD-ROM. Type the following line in the Message window to add the text between the quotation marks to the field `myField`:

```
put "Too soon old, too late smart" into field "myField"
```

Note that field `myField` here refers to cast member 1. You can use this command to put text into an empty field or text member or to replace existing text. However, if member 1 is something other than a field — such as a text, bitmap, sound, or script cast member — the instruction does not work and the contents of the cast member remain unchanged. The empty member is on the Stage, so when the command is executed, you see the text appear on the Stage and in the member. For more information on the effects of changing the content of a member, see the sidebar "When Is the Stage Updated?"

You can use a slightly different form of the `put` command to place text in a field cast member:

```
put "Too soon old, too late smart" into member "myField"
```

This applies whether you use a cast member name or a number in the syntax of the command.

You can use a similar syntax to place a string into either a local or global variable:

```
put "You can't tell how deep a puddle is until you step into
it." into proverbs
```

To initialize a variable (that is, to establish what the variable represents before any processing occurs), you can use an empty string. You do this by placing an empty string in a cast member with either of the following commands:

```
put "" into field 1
put EMPTY into member 1
```

On the CD-ROM

The text3.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

When is the stage updated?

Changes to sprite properties, such as the locH, locV, or the ink, take place when the playback head moves (to another frame or back to the same frame it is on) and when the `updateStage` command is used.

A field is updated as soon as it changes, even in a repeat loop. Changed text members and vector shapes are updated when sprite properties are updated, unless a field is being updated. If a field has its contents changed (and it is on the Stage), text members and vector shapes also are updated if their content has been changed—instead of waiting for an `updateStage`! To see this in action, look at text3.dir on the CD-ROM.

Play the movie text3.dir. You see the text change in both the field at the top and the text member at the bottom. You also see the four-sided vector shape change its shape when the text changes. After a few seconds, you see the vector shape change locations. If you look in the code, you see that the location should change every time the text changes, for a total of five times each `exitFrame`. It is only changing once, because the sprite's properties are only updated upon entering the frame (unless you use the `updateStage` command). If you uncomment the `updateStage` command in the `ChangeLoc` handler, you will see the shape move whenever the text changes.

Caution

If you removed the `updateStage` and the field member from the Score (or just commented the line of code that alters the field's contents), you will *not* see the text member or vector shape change in the repeat loop! This is an oddity of Director fields. Knowing this can save you a long and difficult bug hunt if you ever encounter text or vector shapes being updated mysteriously.

EMPTY is a character constant and always has the same value: an empty string. Character constants are not case sensitive, so the constant can be entered as **EMPTY**, **Empty**, or **empty**.

When referring to a field, you can use its name or cast member number. If you use a field name, you must enclose it within quotation marks. If you use a number, Director assumes you are referring to a member in castLib 1, the internal castLib. For example, if you want to put text into a field `myField` that is in castLib 2 and not the field named `myField` in castLib 1, you could write it one of several ways:

```
put "Easy come, easy go" into field "myField" of castLib 2
put "Easy come, easy go" into member "myField" of castLib 2
put "Easy come, easy go" into member("myField", 2)
```

In any of the preceding, you can reference the castLib by name, as follows:

```
put "Easy come, easy go" into member("myField", "Fields")
```

It is better to use a cast member's name, rather than its number. When you reference a member by name, Director searches all of the castLibs until it finds one that matches. This enables you to move your cast members around without worrying about changing the code. Referencing them by name also makes the code more readable. Remember to surround the name in quotation marks; otherwise, Director thinks you are trying to use a variable.

Lingo uses quotation marks to identify the beginning and end of a string. To include quotation marks within a string, you must use the QUOTE constant, as shown in the following example:

```
put "T. S. Eliot once said," && QUOTE & "Only those who risk
going too far can possibly find out how far one can go." &
QUOTE into member "myText"
```

If the field or text member into which you place a string is not on the Stage, it is stored in the member, but it is not displayed on the Stage. This might seem obvious, but it is useful to point out because you can change a string before it is displayed in your movie.

Understanding chunk keywords

You can use chunk expressions to select a portion of a string (a chunk of it). You can use chunk expressions with fields, text members, and variables containing strings. Table 16-1 shows the different levels of chunk expressions.

Table 16-1
Chunking Levels

Level	*Keyword*	*Example*
`Characters`	`char`	`put member("field_or_text").char[5]` `put char 5 of field "myField"`
`Words`	`word`	`put member("field_or_text").word[2]` `put word 2 of field "myField"`
`Items`	`item`	`put member("field_or_text").item[3]` `put item 3 of field "myField"`
`Lines`	`line`	`put member("field_or_text").line[2]` `put line 2 of field "myField"`
`Paragraphs`	`paragraph`	`put member("field_or_text").paragraph[1]`

Using the char keyword

You use the `char` keyword to get or set one or more characters in a text member, field, or string.

On the CD-ROM

The text4.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Getting a char

Open the text4.dir movie on the CD-ROM. In the Message window, type the following:

```
put field "myField" into aQuote
put aQuote
-- "One for the money,
two for the show,
three to get ready,
and four to go."
```

The variable `aQuote` now holds a copy of the text that is in the field `myField`. To get a single `char` from the string variable `aQuote`, type the following:

```
put char 3 of aQuote
-- "e"
```

To get a range of characters, type this:

```
put char 1 to 10 of aQuote
-- "One for th"
```

This same syntax works for fields. In the place of the string (or string variable), you use a field reference, as follows:

```
put char 2 of field "myField"
-- "n"
```

If you try to use a member reference instead, as in the following, you get an error:

```
-- THIS SYNTAX DOESN'T WORK!
put char 5 of member "myField"
```

Director considers the syntax field `myField` to be a string, so string operations work on it. You can test this by using the `stringP` function to test whether something is a string:

```
put stringP(field "myField")
-- 1
put stringP(member "myField")
-- 0
```

Using the `field` keyword returns a string, so it tests true (1). Using the `member` keyword returns a reference to a member, so `stringP` returns false (0).

To access text in a text member, you need to use a variation on the syntax or use the dot notation, as shown here:

```
put char 10 of the text of member "myText"
-- "h"
```

Note the addition of the `text` keyword. This syntax also works with fields. The problem with the syntax is that you cannot use a similar syntax for setting `chars`. This is because the `text` keyword returns the text of the member, so you are not manipulating the text in the member. It is better to use the new dot notation:

```
put member("myText").char[10]
-- "h"
```

You can use this method on fields by using the `member` keyword and on string variables, but not string literals:

```
put member("myField").char[6]
put field("myField").char[6]
put aQuote.char[6]
```

Getting a range of chars

To get range of chars, use this syntax:

```
put char 1 to 3 of field("myField")
-- "One"
put char 21 to 23 of aQuote
-- "two"
```

This syntax does not work with text members. To access a range of `char`s from a text member, it is necessary to use the dot notation, as shown here:

```
put member("myText").char[40..44]
-- "three"
```

This syntax works with strings and fields as well.

Setting a char

To set a `char`, you use the `put. . .into` syntax for fields and strings:

```
put "o" into char 1 of field "myField"
put "o" into char 1 of member "myField"
put "o" into char 1 of aQuote

put "o" into char 1 of member "myText" -- DOES NOT WORK!
```

For text members, use the dot notation with the assignment operator (equals sign), as follows:

```
member("myText").char[5] = "X"
```

Just because you are setting the value of one `char`, it doesn't mean you cannot insert more:

```
member("myText").char[5] = "XYZ"
```

Unfortunately, the dot notation does not work when assigning values to strings or fields:

```
field("myField").char[1] = "X"  -- DOES NOT WORK!
aQuote.char[1]= "Y" -- DOES NOT WORK!
```

The assignment of `chars` with dot notation does work with fields if you use the `member` keyword, as shown here:

```
  member("myField") = "Z" -- WORKS!
```

Setting a range of chars

Setting a range of characters is different depending on whether it is a string, field, or text member and the notation that you use — either standard or dot:

```
  put "1" into char 1 to 3 of aQuote
  put "2" into char 21 to 23 of field "myField"
  put "One" into char 1 to 3 of member "myField"
  put "XXX" into char 1 to 3 of member "myText" -- DOES NOT WORK
```

You need to use dot notation to set text in a text member. But you'll find it does not work for everything. The same restrictions for setting a single `char` exist for a range of `chars`:

```
  aQuote.char[1..3] = "1" -- DOES NOT WORK
  field ("myField").char[5..8] = "Two" -- DOES NOT WORK!
  member("myField").word[9..12] = "Three"
  member("myText").word[13..16] = "Four"
```

Counting chars

Counting chars is easy, but don't use the `count()` function from lists! `Chars` use the `number` keyword, as shown here

```
  put the number of chars in aQuote
  -- 75
  put the number of chars in field "myField"
  -- 75
  put the number of chars in member "myText" -- DOES NOT WORK
```

To get the number of `chars` using the `member` keyword with either fields or text, you need to use the `text of member` syntax:

```
  put the number of chars in the text of member "myText"
  -- 75
```

For strings and fields, you can use the `length()` function:

```
  put length(aQuote)
  -- 75
  put aQuote.length
  -- 75
  put length(field "myField")
  -- 75
```

Here is another odd exception:

```
put aQuote.length() -- DOES NOT WORK
put field("myField").length() -- DOES NOT WORK
```

The first one does not work because parentheses were added, which is an inconsistency in Lingo. Usually, if you are calling a function, you need parentheses. As in the `count()` function syntax for lists — `myList.count()`. If you were to forget to add the parentheses, you would get an error. Oddly, the dot notation does not work with parentheses after `length`. Hopefully, inconsistencies like these will change in a future update.

Using the word keyword

The word *chunk expression* is used in a fashion similar the use of `char`. The way Director determines a word is based on the space, Return, and Tab characters. Any characters between those — and at the beginning and end of a string — is a word. The result does not include those characters. It can be said that the word command uses the space, Return, and Tab characters as a *delimiter* (a delimiter is a character that is used to separate chunks of text). Given the similarities between the syntax for `char`s and the remaining chunk expressions, the examples are shorter.

Getting words

If you've made changes to text4.dir, just choose File ➪ Revert to start fresh. If you've changed the variable `aQuote`, reset it now by typing the following:

```
put field ("myField") into aQuote
```

Now, try putting some of the words to the Message window from the string, field, and text, as follows:

```
put word 1 of aQuote
-- "One"
put word 2 of field "myField"
-- "for"
put word 3 of member "myText" -- DOESN'T WORK!
put word 3 of the text of member "myText"
-- "the"
```

Once again, the `member` keyword trips us up when using a chunk expression! Now try the dot notation:

```
put aQuote.word[4]
-- "money,"
put field("myField").word[5] --DOESN'T WORK!
put member("myField").word[5]
-- "two"
put member("myText").word[6]
-- "for"
```

Arrrgghh! This time, the `field` keyword tripped us up. Given the subtle differences between using fields as opposed to text members, you can see why it might be easier to retain your sanity by sticking to just one or the other. One nice aspect of fields is that the syntax is similar to that of strings. Still, text member capabilities seem to outweigh that advantage. So, you are forced to remember both the string and text syntax.

Getting a range of words

Getting a range of words is just like getting a range of `char`s, which you do as follows:

```
put word 1 to 4 of aQuote
-- "One for the money,"
put word 5 to 8 of field ("myField")
-- "two for the show,"
put word 9 to 12 of member ("myField")
-- "three to get ready,"
put word 13 to 16 of the text of member ("myText")
-- "and four to go."
```

Note that you can use the keywords `field` or `member` for fields. `the text of member` is added to get the words from `"myText"`. The following code example uses the dot notation. Note that to get the words from `"myField"`, you need to use the `member` keyword. If you use the `field` keyword with dot notation, you get unexpected results:

```
put aQuote.word[1..4]
-- "One for the money,"
put member("myField").word[5..8]
-- "two for the show,"
put member("myText").word[9..12]
-- "three to get ready,"
```

Setting a word

Setting a word has its idiosyncrasies as well, as shown here:

```
put "1" into word 1 of aQuote
put "2" into word 5 of field "myField"
put "3" into word 9 of member "myField"
put "4" into word 13 of member "myText" -- DOES NOT WORK!
```

The only way to set a word with a text member is with dot notation. Even the `text of member` does not work when setting a word in a text member:

```
aQuote.word[1] = "One"  -- DOES NOT WORK!
field ("myField").word[5] = "Two" -- DOES NOT WORK!
member("myField").word[9] = "Three"
member("myText").word[13] = "Four"
```

Setting a range of words

Setting a range of words is like setting a range of `chars`:

```
  put "Testing" into word 1 to 3 of aQuote
  put "1,2,3" into word 21 to 23 of field "myField"
  put "One" into word 1 to 3 of member "myField"
  member("myField").word[9..12] = "Three"
  member("myText").word[13..16] = "Four"
```

Counting words

Counting words works like counting `chars`, except you don't have the option of using the `length` function, as in the following:

```
  put the number of words in aQuote
  -- 16
  put the number of words in field "myField"
  -- 16
  put the number of words in the text of member "myText"
  -- 16
```

Chars of words

You can use smaller chunk expressions, such as `char`, with larger chunk expressions, such as `word`, as shown here:

```
  put char 1 of word 2 of the text of member("myText")
  -- "f"
```

The equivalent dot notation is a bit less verbose:

```
  put member("myText").word[2].char[1]
  -- "f"
```

Using the item keyword

The `item` chunk expression retrieves one or more strings of characters that are delimited (separated) by commas in a text string. If you are familiar with database structures, you might have heard of comma-delimited fields. In Director, comma-delimited fields are considered items. Note that the separator (in this case, the comma character) is not considered part of the item.

Getting items

Getting items is just like getting words, except the delimiter is a comma by default:

```
  put item 1 of field "myField"
  -- "One for the money"
  put member("myField").item[2]
  -- "
  two for the show"
```

As mentioned earlier for the other chunk expressions, you cannot use the dot notation with the `field` keyword, but you can use the standard notation or use the `member` keyword with a field. Note that the RETURN character was included when you got item 2. The RETURN came right after the first comma, so it is considered part of the second item. One nice aspect of getting items is that you can change the delimiter. The delimiter is kept in the system property, `the itemDelimiter`.

By setting the delimiter to a RETURN, you can get items in a fashion similar to the `line` keyword:

```
the itemDelimiter = RETURN
put member("myField").item[1]
-- "One for the money, "
put member("myField").item[2]
-- "two for the show, "
```

By setting the delimiter to a SPACE, you can have it act in a fashion similar to the `word` keyword, as follows:

```
the itemDelimiter = SPACE
put member("myField").item[1]
-- "One"
```

Caution

When setting `the itemDelimiter`, it is a good practice to save the current value in a variable before changing it. When you are finished with getting items, you restore `the itemDelimiter` to its previous value (held in the variable), if necessary.

Setting items

Setting an `item` has the same limitations as setting a `word`:

```
put "1" into item 1 of aQuote
put "2" into item 5 of field "myField"
put "3" into item 9 of member "myField"
put "4" into item 13 of member "myText" -- DOES NOT WORK!
```

The only way to set a `word` with a text member is with dot notation. Even the text of member does not work when setting a `word` in a text member:

```
aQuote.item[1] = "One"  -- DOES NOT WORK!
field ("myField").item[5] = "Two" -- DOES NOT WORK!
member("myField").item[9] = "Three"
member("myText").item[13] = "Four"
```

Setting a range of words

Setting a range of `words` is like setting a range of `chars`:

```
put "Testing" into item 1 to 3 of aQuote
put "1,2,3" into item 21 to 23 of field "myField"
```

```
  put "One" into item 1 to 3 of member "myField"
  member("myField").item[9..12] = "Three"
  member("myText").item[13..16] = "Four"
```

Counting items

Counting items works like counting `words`:

```
  put the number of items in aQuote
  -- 4
  put the number of items in field "myField"
  -- 4
  put the number of items in the text of member "myText"
  -- 4
```

Chars of words of items

Just as you could use `char` with `word`, you can use both of those with `item`, as in the following example:

```
  put char 1 of word 1 of item 2 of the text of member("myText")
  -- "t"
```

The equivalent dot notation is a bit less verbose:

```
  put member("myText").item[2].word[1].char[1]
  -- "t"
```

When doing a range of `chars` or `words`, it is possible for the larger chunk expression to be a part of the range of the smaller, such as in this example:

```
  put member("myText").word[2..6].item[2].char[3]
  -- "t"
```

To understand what is happening, break it down into more manageable expressions. Here is the range of `words`:

```
  put member("myText").word[2..6]
  -- "for the money,
  two for"
```

Given that result, the second item is obvious (assuming the `itemDelimiter` is set to a comma):

```
  put member("myText").word[2..6].item[2]
  -- "
  two for"
```

The third character is “t”; it is right after the SPACE and RETURN characters.

Using the line keyword

The line chunk expression retrieves a string of characters that is preceded and followed by a RETURN character in a text string. Note that the separator (such as the RETURN character) is not considered part of the line. Because the line keyword uses a RETURN character to delimit between chunks, it really acts as if it is getting paragraphs. A line might appear on the screen as multiple lines because the line wraps.

Using the paragraph keyword

The paragraph chunk expression works like the line chunk expression, but it only works with dot notation. The paragraph keyword was a new addition to version 7. It seems like a better-named version of the line keyword.

Using the ref keyword

The ref keyword is a chunk expression that only works with text cast members. It enables you to create a shorthand way of referring to a chunk of text. Instead of writing the following:

```
member("myText").line[2].word[2..3]
```

You can assign a reference to that chunk by doing the following:

```
myRef = member("myText").line[2].word[2..3].ref
```

Note the addition of .ref at the end. myRef now contains a reference to that chunk of text:

```
put myRef
-- <Prop Ref 2 31d9004>
```

Now you can get and set properties of just that chunk of text, as shown in the following:

```
put myRef.text
-- "for the"
put the fontStyle of myRef
-- [#plain]
put myRef.font
-- "Hanzel Condensed-Bold *"
myRef.fontStyle = [#italic]
myRef.color = paletteIndex(5)
```

Hyperlinks in Text Cast Members

Director has the capability to embed hyperlinks in text members. You can import these hyperlinks from an HTML document, or you can create them in Director.

On the CD-ROM

The hyperlinks.dir movie is in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Open the hyperlinks.dir movie on the CD-ROM. Use this movie for the code samples in this section.

hyperLink

This property returns the `hyperlink` string for a chunk expression. You can get or set this property, but you must use dot notation. To clear a hyperlink, set it to `EMPTY`, or `""`:

```
  put member("myText").word[2].hyperlink
  -- "http://www.marsproject.com/"
```

A hyperlink can contain any string, not just URLs:

```
  member("myText").word[21].hyperlink = "go to the frame + 1"
```

hyperlinks

This text cast member property returns a list of hyperlink ranges. Each range is a list containing two integers. The first item in the range is the starting character, and the second item is the ending character. You cannot set this property:

```
  put the hyperlinks of member "myText"
  -- [[5, 25], [107, 116]]
  put member("myText").hyperLinks
  -- [[5, 25], [107, 116]]
```

hyperlinkRange

This property returns the range of the hyperlink in the chunk expression. The first character in the chunk expression must be part of the hyperlink. You cannot set this property:

```
  put member("myText").word[21].hyperlinkrange
  -- [107, 116]
```

hyperlinkState

This property contains the current state of the hyperlink. You can get or set this value. Possible values are `#active`, `#normal`, and `#visited`:

```
put member("myText").word[2].hyperlinkState
-- #normal
member("myText").word[2].hyperlinkState = #visited
```

Other Chunk Expressions

There are several other chunk expressions that are useful in Lingo.

last()

This function retrieves the `last` chunk expression of the string. It can be used with `char`, `word`, `item`, and `line`:

```
put the last item of "Easy come, easy go."
-- " easy go."
put the last word of "Easy come, easy go."
-- "go."
put the last char of "Easy come, easy go."
-- "."
```

contains

You've seen this function used already if you've been reading sequentially. It tells you whether one string contains another. The result is either 1 or 0:

```
put "All good things..." contains "good"
-- 1
```

This is one of those times where the Lingo code reads nicely.

offset()

This function returns an integer that is the position of the first occurrence of a substring in another string:

```
put offset("good", "All good things...")
-- 5
```

`"g"` is the fifth character in the string `"All good things. . ."`. If the substring is not in the string, 0 is returned. To use this with a text member, remember to use the `text` keyword:

```
put offset("the", member("myText").text)
-- 9
```

Formatting Paragraphs in Text Members

Director has many properties for formatting paragraphs. Only text members have these properties (with the exception of the `alignment` property), but because that is the text format you are most likely to use, it is not much of a limitation. Anyone familiar with page layout programs will find these properties familiar and welcome.

With these commands, you can specify a different value for each paragraph. The syntax is as follows:

```
member("myText").paragraph[n].propertyName = aValue
```

`n` is the number of the paragraph you're accessing. `propertyName` is the property you are setting. `aValue` is the value you are giving that property.

If you want to set the entire paragraph in a text member to the same value, use the same syntax, but leave off the `paragraph` keyword:

```
member("myText").propertyName = aValue
```

fixedLineSpace

This is analogous to the field command `lineHeight`. Technically, it can be used with chunk expressions other than just paragraphs, but it falls logically into this category. In typesetting, this would be called *leading*. The default is 0, which means it spaces relative to the point size. Any other value sets the spacing to that value regardless of point size.

On the CD-ROM

You can find the text6.dir movie in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

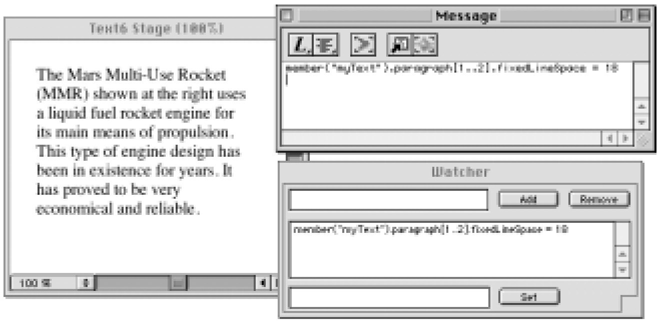
Open the text6.dir movie on the CD-ROM. Change the `fixedLineSpace` of member `myText`, as shown in Figure 16-3, by typing the following in the Message window:

```
member("myText").paragraph[1..2].fixedLineSpace = 18
```

The Watcher window is useful for seeing the results when you change values of an object. Enter the expression `member("myText").paragraph[1..2].fixedLineSpace` into the Watcher window.

Note

You do not enter the `= 18` that appears automatically to show the value of the expression. The Watcher window shows you what an expression evaluates to. In the Message window, you are typing an executable line of Lingo. When you use the equals sign (=) in the Message window, it is acting as the assignment operator, assigning 18 to the `fixedLineSpace` property of paragraphs 1 through 2 of the member `myText`. In the Watcher window, it is just acting as an equals sign, showing you what the expression equals (evaluates to).



The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion. This type of engine design has been in existence for years. It has proved to be very economical and reliable.

```
member("myText").paragraph[1..2].fixedLineSpace = 18
```

```
member("myText").paragraph[1..2].fixedLineSpace = 18
```

Figure 16-3: Setting the fixedLineSpace and viewing the results in the Watcher window

bottomSpacing and topSpacing

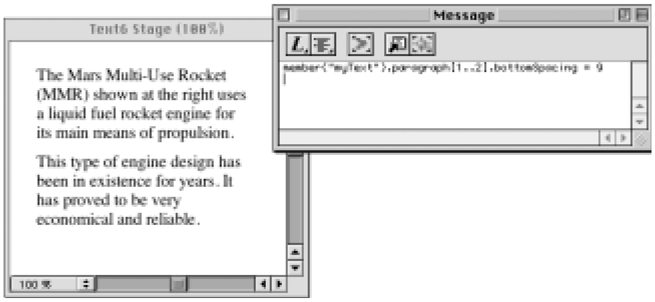
A paragraph might wrap and form many lines. Sometimes you want the space between paragraphs to be different than the space between lines. You can use `bottomSpacing` or `topSpacing`, or both, to achieve this. It is probably best to pick the one that makes sense to you and stick with just that one, as opposed to sometimes using `topSpacing` and sometimes `bottomSpacing`.

We might be inclined to prefer `bottomSpacing` with Director for an important reason: When you add `topSpacing` to the first paragraph in the member, it pushes the text away from the top of the member. While this is perfectly logical, programs such as QuarkXPress and Adobe Illustrator ignore the `topSpacing` when it is the first paragraph in a text box. Oddly enough, Director does not add space to the bottom of the text member, and it does not push the rect of the member farther down when you increase the `bottomSpacing` on the last paragraph.

To get or set the `topSpacing` or `bottomSpacing`, you do it the same way you would with other member properties:

```
  member("myText").paragraph[1].bottomSpacing = 9
```

If paragraph 1 had a `fixedLineSpace` of 18, then the space between the baseline of the last line in paragraph 1 and the first line of paragraph 2 is 27 pixels (18 + 9). The result is shown in Figure 16-4. By using `bottomSpacing` (or `topSpacing`), you can space paragraphs to your liking, instead of using a hard return or separate text members.



The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion.

This type of engine design has been in existence for years. It has proved to be very economical and reliable.

```
member("myText").paragraph[1..2].bottomSpacing = 9
```

Figure 16-4: The result of setting the bottomSpacing of paragraphs 1 to 9

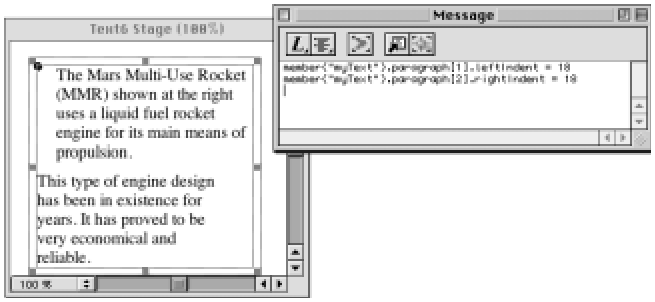
leftIndent and rightIndent

You change the left and right indents by setting the `leftIndent` and `rightIndent` properties. The `leftIndent` is how much the entire paragraph is indented on the left side of the text cast member. The `rightIndent` is how much the text is indented from the right side of the member, as in this example:

```
member("myText").paragraph[1].leftIndent = 18
```

In the preceding example, the text for paragraph 1 would now be indented 18 pixels. Figure 16-5 shows the result; it also illustrates the `rightIndent` set to 18 in paragraph 2. To set paragraph 2, type the following code:

```
member("myText").paragraph[2].rightIndent = 18
```



The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion.

This type of engine design has been in existence for years. It has proved to be very economical and reliable.

```
member("myText").paragraph[1].leftIndent = 18
member("myText").paragraph[2].rightIndent = 18
```

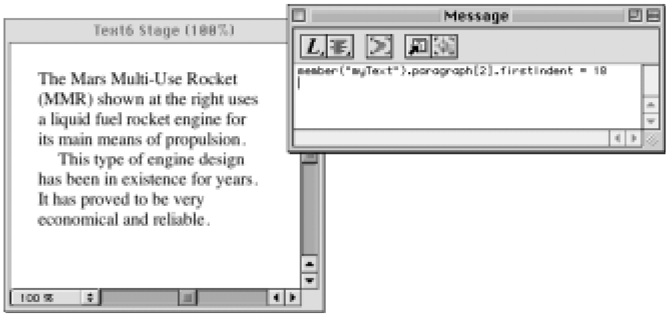
Figure 16-5: An example of setting the leftIndent in paragraph 1 and the rightIndent in paragraph 2

firstIndent

To set the indent for the first line of a paragraph, you set the `firstIndent` property (Lingo names can be pretty intuitive sometimes, can't they?), as in this example:

```
  member("myText").paragraph[2].firstIndent = 18
```

You can set the `firstIndent` to a negative value up to but not greater than the positive value of the `leftIndent`. In other words, if the `leftIndent` of a paragraph is 0, then a negative value for `firstIndent` has no effect — not even an error message — it's just ignored. If the `leftIndent` is set to 18, then the `firstIndent` can be up to a –18. This provides for "outdent" or "hanging indent" paragraph formats. A useful application for `firstIndent` is bullet points. Typically, bullet points should be outdented from the paragraph's left margin. The preceding indent would be 18 pixels in from whatever the left indent is. Figure 16-6 shows paragraph 2 with the `firstIndent` set to 18. This is another way to distinguish subsequent paragraphs, instead of increasing the `bottomSpacing` or `topSpacing`.



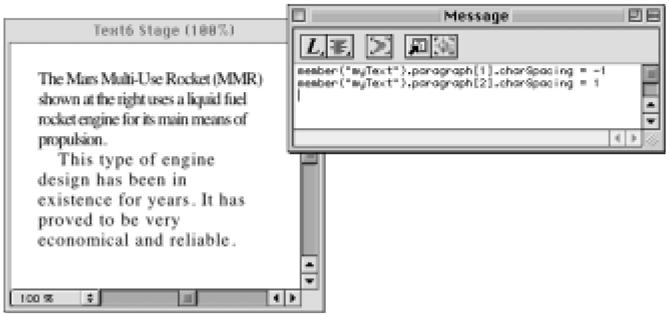
The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion.
    This type of engine design has been in existence for years. It has proved to be very economical and reliable.

```
member("myText").paragraph[2].firstIndent = 18
```

Figure 16-6: Paragraph 2 has its firstIndent set to 18. Paragraph 1's firstIndent is set to 0.

charSpacing

Like the `fixedLineSpace` property, this property is not exclusive to paragraphs, but it logically fits into this category. This property can be set to positive or negative values, and the default is 0. Figure 16-7 shows an example of `charSpacing`.



The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion.

This type of engine design has been in existence for years. It has proved to be very economical and reliable.

```
member("myText").paragraph[1].charSpacing = -1
member("myText").paragraph[2].charSpacing = 1
```

Figure 16-7: Paragraph 1 has a charSpacing of –1; paragraph 2 has a charSpacing of 1.

tabCount

This property indicates the number of tab stops in a text member. It can be tested, but it cannot be set.

tabs

This property is a list of all the tabs in a text member. It is a linear list of property lists. Each property list has the properties `#type` and `#position`. The values for `#type` can be `#left`, `#right`, `#center`, or `#decimal`. `#position` is an integer indicating the position of the tab. You can get and set this property.

alignment

Of the paragraph commands, this is the only one that also works with fields, but not in the same way. With a text member, you can set the alignment of any paragraph in the member. The value you assign the alignment property must be a symbol:

```
  member("myText").paragraph[2..3].alignment = #right
```

Fields do not enable you to set individual formatting of paragraphs, so for fields you must use the preceding syntax, with one important change — the value assigned must be string, as in this example:

```
  member("myField").alignment = "left"
```

Scrolling Text

Fields and text members that have been turned into scrolling text boxes have their own set of Lingo commands, which are described in the following sections.

scrollByLine

This command takes two arguments, the member and an integer. It scrolls a member up or down by the number of lines specified by the integer. A positive amount scrolls down; a negative amount scrolls up:

```
scrollByLine member "text_or_field", 2
```

scrollByPage

This command takes two arguments, the member and an integer. It scrolls a member up or down a page at a time. A *page* is the number of lines in the member that are visible. A positive amount scrolls down; a negative amount scrolls up. The syntax is:

```
scrollByLine member "text_or_field", -1
```

scrollTop

The `scrollTop` is the amount of the cast member scrolled off the top of the member. By setting it to 0, you can make sure the first line of the member is at the top of the member:

```
member("text_or_field").scrollTop = 10
```

Getting Line and Char Locations

Lingo provides many terms for determining the location of the mouse in relation to the text on the screen. The first four terms use locations relative to the upper-left corner of the text box, and the remaining terms use locations relative to the Stage.

linePosToLocV

This function returns a line of text's distance from the top of the member; this value is relative to the member, not to the position of a sprite on the Stage:

```
linePosToLocV(member "text_or_field", 3)
```

If the preceding code evaluated to 24, and the sprite's locV was 100, line 3 would be at 124 pixels from the top of the Stage.

locVToLinePos

This is the complement of `linePosToLocV`; it returns a number that indicates which line is closest to the locV you pass in as an argument to the function. The locV is relative to the top of the member, not to the position of a sprite on the Stage:

```
locVToLinePos(member "text_or_field", 24)
```

If the preceding code evaluates to 3, then line 3 of the member is closest to 24 pixels down from the top of the member.

locToCharPos

This function returns an integer, which is the number of the character in a field or text member that is closest to the point passed in:

```
locToCharPos(member "text_or_field", point(95, 150)
```

If the preceding code returned 10, then that would mean character 10 in the text of the field or text member was closest to `point(95, 150)`.

Caution The point you pass as an argument in this function is relative to the location of the upper left corner of the sprite.

charPosToLoc

This function is the complement of `locToCharPos`. It returns the position of the character relative to the upper-left corner of the member:

```
charPosToLoc(member "text_or_field", 10)
```

If the preceding code returns points (95, 150), you would need to add the left and top values of the rect of the sprite the text is in to get the location of the character on the Stage.

The pointTo functions

The remaining terms are `pointToChar`, `pointToWord`, `pointToItem`, `pointToLine`, `pointToParagraph`, and `pointToHyperLink`. As you might have guessed from their names, all of the functions correspond to the chunk expressions: `char`, `word`, `item`, `line`, and `paragraph`. These functions are similar to `locToCharPos`, except the point you pass these functions is not relative to the upper-left corner of the member but to a screen coordinate. The return value is the number of the chunk expression the point is over:

```
pointToChar(member "text", the mouseLoc)
```

In the preceding code, if the mouse is over the member, it returns an integer indicating which character. If it is not over any character, it returns –1. Because you use Stage coordinates, these functions are easier to work with, but they work only with text members.

More Text Cast Member Properties

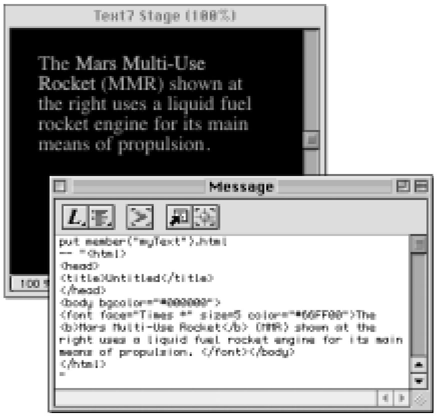
The number of text member properties might seem overwhelming, but there are still several more properties that are important and worth discussing.

HTML and text members

Not only can Director import HTML files (with some limitations), but every text member has an HTML property that you can get and set.

On the CD-ROM You can find the text7.dir movie in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Open the text7.dir movie on the CD-ROM. In the Message window, type `put member("myText").HTML`, as shown in Figure 16-8, or `put the HTML of member("myText")`.



The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion.

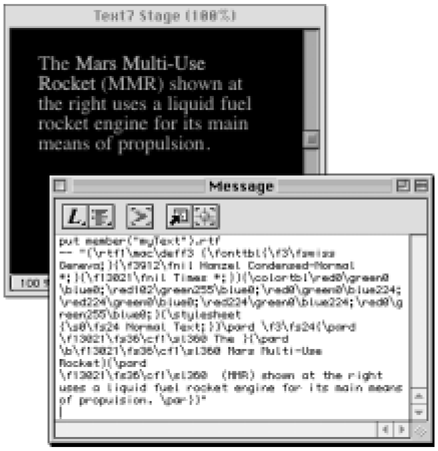
```
put member("myText").html
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#000000">
<font face="Times *" size=5 color="#66FF00">The
<b>Mars Multi-Use Rocket</b> (MMR) shown at the
right uses a liquid fuel rocket engine for its main
means of propulsion. </font></body>
</html>
"
```

Figure 16-8: Every text member has an HTML property that can be retrieved or changed.

Note how the font, the size, and the color are specified in the font tag. Within the text, the bold words **Mars Multi-Use Rocket** correctly have the `<b></b>` tags.

RTF

Besides having an HTML property, each text member has an RTF property as well. If you are familiar with the Rich Text Format (RTF), you can get and set this value as well. Figure 16-9 shows the result of getting the RTF of a member.


Text7 Stage (100%)
The Mars Multi-Use Rocket (MMR) shown at the right uses a liquid fuel rocket engine for its main means of propulsion.
Message

```
put member("myText").rtf
-- "{\rtf1\mac\deff3 {\fonttbl{\f3\fswiss
Geneva;}{\f3912\fnil Hanzel Condensed-Normal
*;}{\f13021\fnil Times *;}}{\colortbl\red0\green0
\blue0;\red102\green255\blue0;\red0\green0\blue224;
\red224\green0\blue0;\red224\green0\blue224;\red0\g
reen255\blue0;}{\stylesheet
{\s0\fs24 Normal Text;}}\pard \f3\fs24{\pard
\f13021\fs36\cf1\sl360 The }{\pard
\b\f13021\fs36\cf1\sl360 Mars Multi-Use
Rocket}{\pard
\f13021\fs36\cf1\sl360  (MMR) shown at the right
uses a liquid fuel rocket engine for its main means
of propulsion. \par}}"
```

Figure 16-9: Every text member has an RTF property that can be retrieved or changed.

antiAlias and antiAliasThreshold

Text members antialias text by default. By default, the threshold at which the anti-aliasing occurs is at 14-point and larger type. To turn off antialiasing for a particular member, just set the `antiAlias` property to FALSE. To turn antialiasing back on, set the `antiAlias` property to TRUE. Rendering speed of the sprites is slower with antialiasing turned on. Following are examples of dot and standard notation for turning the antialiasing off and then on again for member `myText`:

```
member("myText").antiAlias = FALSE
set the antiAlias of member "myText" to TRUE
```

If anti-aliasing is desired at a different size, change the `antiAliasThreshold`. Setting it to 0 makes all text sizes antialiased. Antialiasing small text can sometimes make readability worse; it varies from font to font. Following are examples of dot and standard notation for setting the `antiAliasThreshold` property:

```
member("myText").antiAliasThreshold = 18
set the antiAliasThreshold of member "myText" to 18
```

kerning and kerningThreshold

To have the text in a member automatically kerned, set the kerning property to TRUE. Setting it to FALSE turns kerning off. Following are examples of dot and standard notation for turning the kerning off and then on again for member `myText`:

```
member("myText").kerning = FALSE
set the kerning of member "myText" to TRUE
```

The `kerningThreshold` property determines the point size at which type is kerned in a member. The default is 14. Following are examples of dot and standard notation for setting the `kerningThreshold` property:

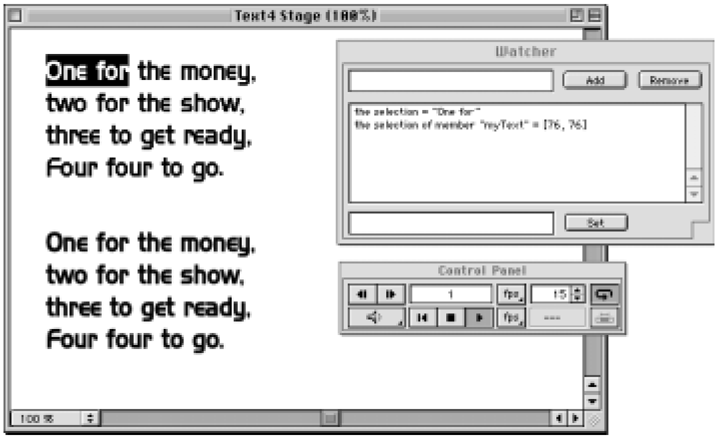
```
member("myText").kerningThreshold = 21
set the kerningThreshold of member "myText" to 21
```

Selecting text

Selecting text is done two different ways: one way for fields and one way for text members.

the selection

This keyword might be a little confusing because of how different it is between fields and text members. `the selection` without a text member reference returns a string of the text selected in a field (see Figure 16-10).



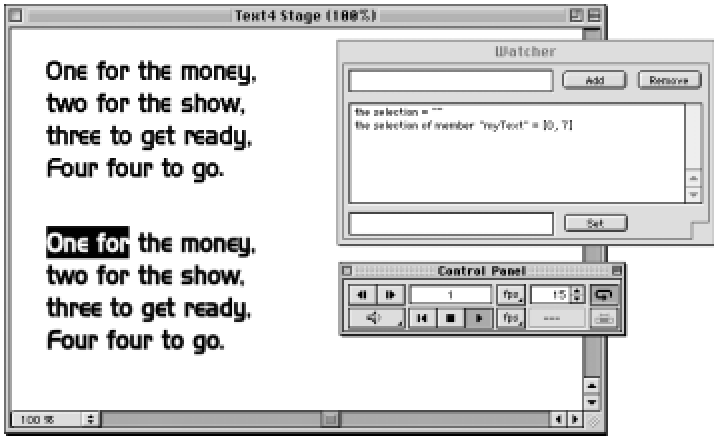
```
the selection = "One for"
the selection of member "myText" = [76, 76]
```

Figure 16-10: "the selection" works with fields and returns a string.

The sprite at the top references a field; the bottom one references a text member. Two expressions have been entered into the Watcher window: `the selection` and `the selection of member "myText"`. `the selection` evaluates to a string of the text selected in the field `myField`. Even if there were multiple fields on the screen, text can only be selected in one field at a time. `the selection of member "myText"` evaluates to a list with two integers.

If you use `the selection` with a text member, it returns a list of two integers, as shown in Figure 16-11. The first is the first `char` selected, and the second is the last `char` selected. If the first is greater than the last (as in Figure 16-10), then no `char` is selected, but it does give you an idea of the position of the insertion point. This value can be retrieved even when the member does not have the focus of the keyboard.

When used with text members, you can set the selection.

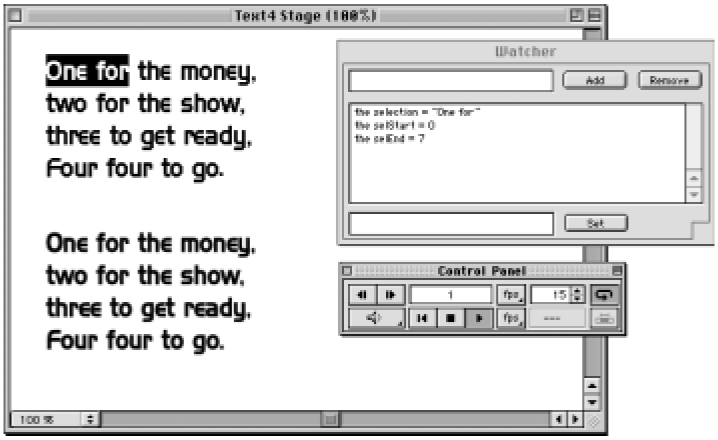


```
the selection = ""
the selection of member "myText" = [0, 7]
```

Figure 16-11: "the selection of member" works with text members and returns a list.

selStart and selEnd

The `selStart` keyword returns an integer that refers to the index of the first character selected in a field. Like the selection, it is not used with a member. The `selEnd` refers to the last character selected. Both work only with fields. If the `selStart = the selEnd`, then nothing is selected, but it does give you an idea of where the insertion point is (see Figure 16-12).



One for the money,
two for the show,
three to get ready,
Four four to go.

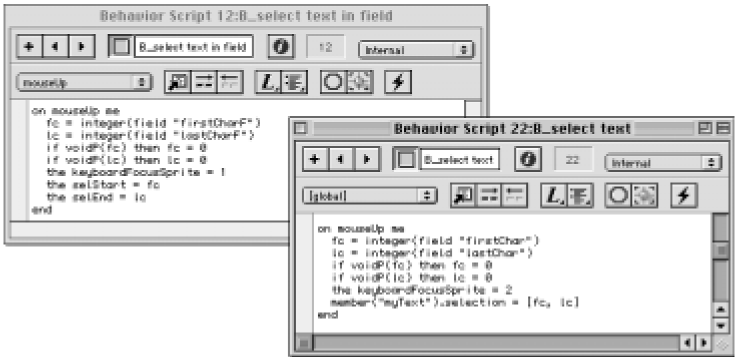
One for the money,
two for the show,
three to get ready,
Four four to go.

```
the selection = "One for"
the selStart = 0
the selEnd = 7
```

Figure 16-12: The selStart and the selEnd work with fields

On the CD-ROM You can find the text5.dir movie in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

To see a movie that has an example of the `selStart`, the `selEnd`, and `the selection of member`, open the text5.dir movie on the CD-ROM. In it, you see two scripts that are very similar, as shown in Figure 16-13.


Behavior Script 12:B_select text in field
B_select text in field
12
Internal
mouseUp
Behavior Script 22:B_select text
B_select text
22
Internal
[global]

```
on mouseUp me
  fc = integer(field "firstCharF")
  lc = integer(field "lastCharF")
  if voidP(fc) then fc = 0
  if voidP(lc) then lc = 0
  the keyboardFocusSprite = 1
  the selStart = fc
  the selEnd = lc
end
```

```
on mouseUp me
  fc = integer(field "firstChar")
  lc = integer(field "lastChar")
  if voidP(fc) then fc = 0
  if voidP(lc) then lc = 0
  the keyboardFocusSprite = 2
  member("myText").selection = [fc, lc]
end
```

Figure 16-13: These scripts for setting the selection in a text member and the selection in a field are very similar.

Both scripts get values from fields used for input and convert them to integers. Then they test to make sure the values are not void. I know it seems strange that a function used to convert something to an integer would return a value that is not an integer (such as `void`), but this is a way for you to test whether the input was valid and then act on it. All that is done here is to set it to zero in the case of `void`. Then the `keyboardFocusSprite` is set to the appropriate sprite. After that is where the differences occur. For a field, you need to set the `selStart` and the `selEnd`. For a text member, you need to specify which text member and then set its selection property to a two-item list. The other important difference is what happens when you set these values. If you set the text member to select `chars` 1 to 7, the first seven characters are selected. If you set the field to select chars 1 to 7, you get `chars` 2 through seven selected. This happens because the `selStart` starts counting with 0. Zero is the point of the selection to the left of the first character and 1 is the point right after it. So, in truth, it is counting based on the area between characters instead of the characters themselves (as in the case with `the selection of member`).

hilite

The `hilite` command works with fields, but it does not work with text members. It enables you to highlight a chunk of text in a specific member, even if it is not an editable field. The syntax can be used with any of the other chunk expression, such as `char`, `word`, `item`, or `line`. To highlight word 2 of the field `myField`, just type the following in the Message window:

```
hilite word 2 of member "myField"
```

selectedText

The `selectedText` works only with text members. It does not return text; it returns a `ref` of the text selected in the text member. This is a handy way to get information on the selection of the user and to give feedback based on the selection.

Field-Only Properties

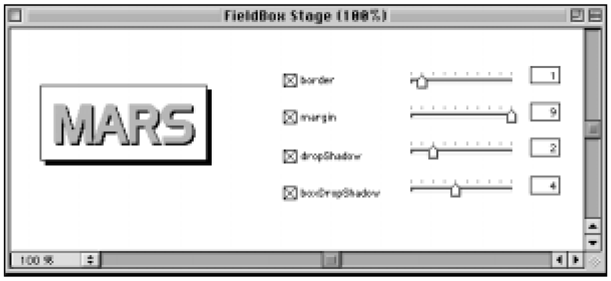
Certain properties apply only to fields such as `hilite`, the `selStart`, and the `selEnd`, which have already been discussed. Perhaps one of the following properties will persuade you to forgo using text cast members.

On the CD-ROM

You can find the fieldbox.dir movie in the EXERCISE:CH16 (EXERCISE\CH16) folder on the CD-ROM.

Setting box properties

You will find fieldbox.dir (see Figure 16-14) on the CD-ROM. Open and play this movie. Click the sliders and slide them back and forth. Each slider corresponds to the check box next to it. Each check box is labeled according to the field property it affects. This is an easy way to see the effects of setting the various properties dynamically. If you deselect any of the boxes, it temporarily sets that value to zero on the MARS field until you select the box again. Even if the box is cleared, you can still move the slider; its effect won't be shown until the check box is highlighted again. The field to the right of the sliders shows the amount to which the property is set. If you were to set these values by choosing Modify ➪ Borders, you would only be able to set the value to a maximum of 5; Lingo enables you to go much higher.



☒ border 1
☒ margin 9
☒ dropShadow 2
☒ boxDropShadow 4

Figure 16-14: The fieldbox.dir movie enables you to play with different settings dynamically.

border

You can create a black border around a field with this command. There is no way of changing the color of the border, so its uses are limited. You can, however, change the width of the border. Figure 16-14 shows a field with a border set to 1. An example of setting it through Lingo is the following:

```
-- dot notation
member("mars").border = 1
-- or standard
set the border of member "mars" to 1
```

margin

This property is the amount of space between the text and the rect of the field. To achieve a similar effect with text members, you would need to modify the `topSpacing`, `leftIndent`, and `rightIndent` properties. Figure 16-14 shows a field with a margin set to 9. An example of setting it through Lingo is as follows:

```
-- dot notation
member("mars").margin = 9
-- or standard
set the margin of member "mars" to 9
```

dropShadow

There is no similar property among text members. This property sets the drop shadow for the text of a field. The default is 0 (no drop shadow). Figure 16-14 shows a field with a `dropShadow` set to 2. An example of setting it through Lingo is as follows:

```
-- dot notation
member("mars").dropShadow = 2
-- or standard
set the dropShadow of member "mars" to 2
```

boxDropShadow

There is no similar property among text members. This property sets the drop shadow for a field. The default is 0 (no drop shadow). Figure 16-14 shows a field with a `boxDropShadow` set to 4. An example of setting it through Lingo is the following:

```
-- dot notation
member("mars").boxDropShadow = 4
-- or standard
set the boxDropShadow of member "mars" to 4
```

Using the wordWrap property

Turning this property off prevents lines from wrapping; only a hard return creates a new line in the field. Text members have nothing similar. An example of setting it through Lingo is this:

```
-- dot notation
member("myField").wordWrap = TRUE
-- or standard
set the wordWrap of member "myField" to TRUE
```

Using the lineCount property

This function is an alternative to using the number of lines in the field `"myField"`. It returns the number of lines in a field. An example of setting it through Lingo is the following:

```
-- dot notation
put member("myField").lineCount
-- or standard
put the lineCount of member "myField"
```

Using the lineHeight property

This is the line spacing for the field (the leading). This property is analogous to the `fixedLineSpace` for text members, except that you can only set the value for the entire field, not for individual paragraphs. An example of setting it through Lingo is this:

```
  -- dot notation
  put member("myField").lineHeight
  -- or standard
  put the lineHeight of member "myField"
```

Using the pageHeight property

This property returns the height of the member that is visible on the Stage, as shown here:

```
  put the pageHeight of member "myField"
```

Summary

Before moving on, recall what you learned in this chapter:

- ✦ Text properties, such as font, size, color, and style, can be controlled through Lingo.
- ✦ Each of these properties can be set for a chunk of text that includes one or more characters, words, lines, paragraphs, or items.
- ✦ There are many subtle differences between fields and text members. Text members give you more options.
- ✦ By using Lingo, you can reset a scrolling text field to the top of the text box, scroll through line by line, and make field text editable.
- ✦ You can create and modify hyperlinks through Lingo.
- ✦ Fields have a few properties that might make them desirable in certain instances.

One of the great things about using text members is their capability to be scaled up or down, because they are resolution independent (unlike bitmaps). In the next chapter, we discuss how to manipulate Director's resolution-independent graphics: vector shape and Flash members (and their associated sprites).

✦ ✦ ✦