MACROMEDIA®

Using Lingo™

Director®

Version 4

First Edition: February 1994
Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

Contents

x

Introduction

Lingo—Director's scripting language that adds interactivity to your multimedia productions—expands the possibilities for your movies. You can create movies that users can explore and travel through in the order that suits them best; that communicate with users by receiving and sending information; that combine animation and sound in ways that the score alone can't; and that let you precisely control text, sound, and digital video.

This introduction to *Using Lingo* tells you:

- The features that Lingo provides
- What this guide contains and what the typographical conventions indicate
- Which Lingo features are new in Director 4.0.

Using Lingo

Lingo adds an intelligent dimension to your movies beyond what the score offers. With Lingo, you can bring the following features to your Director movies:

- *Traveling and exploration*. Your user can choose to travel in and explore different segments of your movie or branch to a separate movie.
- *Interactivity*. You can evaluate and respond to user actions. For example, your movie can let users drag objects to different places on the screen, make choices by clicking buttons, enter text, or ask the user a specific question and let the user know whether the answer is correct.
- *Sprite control*. You can animate sprites and control their location and appearance in response to changing movie conditions and user input.
- *Text manipulation*. You can combine and edit text as the movie plays. In addition to reading text that you display, users can make their own entries part of the text. For example, the movie can prompt users to enter their names and then address the users by name for the rest of the movie.
- *Sound control*. You can play sounds at different times than the score allows, control sound volume, and fade sounds in and out.
- *Working with external files*. You can import data and text files from outside Director and modify them. This lets you build movies that receive and respond to changing input from the outside world.
- *Movies in windows*. Director can play more than one movie at a time. The movies can play independently of each other or interact. For example, you could use a Director movie as the interface for a database.
- *Interfaces*. You can create common interface elements such as menus, buttons, checkboxes, and beeps.

- *Parent scripts*. You can create parent scripts, which you can use to generate a set of objects that share characteristics but can behave independently. For example, Lingo can create a series of buttons that animate in a similar way or a flock of birds that look the same but each fly slightly differently.
- *XObjects*. You can create software modules that interact with external objects such as video cards, and CD-ROM players.
- *Math operations*. You can evaluate many mathematical equations and relay the result for use elsewhere in the movie.
- *HyperCard commands*. You can use XCMDs from HyperCard to perform many functions within Director.

Using this guide

This guide is for the multimedia developer who is already familiar with Director's animation features. It assumes that you've read the manuals *Using Director* and *Learning Director*.

Use this guide to learn about strategies for adding interactivity and flexibility to your movies through Lingo.

Before you start, remember that even though basic Lingo greatly increases your control of a movie, the entire body of Lingo is a deep subject that requires time and effort to learn. Full mastery of Lingo comes from months of experience, but each aspect that you master adds to the ways you can enhance your movies. The more features you know, the more you can achieve; but you don't need to be able to use all features to get useful results.

What's in this guide

This book includes ten chapters, three appendixes, and an index.

Chapter 1, "Script Basics," introduces basic concepts about scripts.

Chapter 2, "Working with Navigation," explains ways to allow movies to branch to different frames or other movies regardless of where frames are placed in the score.

Chapter 3, "Concepts," introduces the theoretical concepts behind Lingo and scripting.

Chapter 4, "Working with Puppets," shows you how to make the contents of the score channels independent of the score and control them directly from Lingo.

Chapter 5, "Manipulating Sprites," shows you new ways to move sprites on the stage, make movie's actions depend on where sprites appear, and change which cast member is assigned to a sprite.

Chapter 6, “Using the Keyboard & Mouse,” describes ways to check what the user enters with the keyboard and mouse and have the movie respond accordingly.

Chapter 7, “Controlling Sound,” shows you how to turn sounds on and off in response to events in the movie instead of from settings in the score, and how to control sound level.

Chapter 8, “Creating Interfaces,” describes ways to create and control interface items such as menus, buttons, and checkboxes.

Chapter 9, “Movies in a Window,” explains how you can play more than one movie at the same time.

Chapter 10, “Parent Scripts and Child Objects,” shows you ways that you can create a set of objects that are similar but can still behave independently of each other.

Appendix A, “Using XCMDs and XFCNs,” describes ways you can use HyperCard XCMDs and XFCNs in your movies.

Appendix B, “Using XObjects,” explains creating and using XObjects.

Appendix C, “Factories,” explains a way to create more than one object from the same Lingo script. This is an alternative to parent scripts.

What you should read

Where you should begin in this guide depends on your background:

- If Lingo is the first scripting language you've worked with, you need to become familiar with some fundamental scripting concepts. Chapter 1, "Script Basics," explains those concepts and presents simple scripts to demonstrate these concepts.
- If you have experience with a scripting language such as HyperTalk but are just learning Lingo, you are already familiar with many of the scripting concepts that Lingo uses and are ready to learn how to write scripts for Director. See Chapter 2, "Working with Navigation," for information about using Lingo to provide navigation within a movie and to branch to other movies.
- If you are familiar with using Lingo for navigating and branching in movies, you can start learning the concepts and features described in Chapters 3 through 10.
- If you are unfamiliar with controlling sprites from Lingo, be sure to read Chapter 4, "Working with Puppets."

Conventions used in this guide

This guide uses the following visual and naming conventions:

- The terms "Lingo" and "Director" refer to version 4.
- Within the text, and in Lingo examples throughout the book, Lingo language elements and parts of actual scripts are shown in `Courier` font.

  Here is a sample line of code:

  ```
  put 2 + 2 into answer
  ```

  Here is a command name within a line of text:

  Use the `set` command to control object properties.

- Quotation marks that are part of Lingo statements are shown in the text and Lingo code examples as straight quotation marks (") rather than as “curly” quotation marks.

  ```
  "This is a quoted Lingo string."
  ```

- Quotation marks that surround names of cast members, movies, or fields are shown in the text as “curly” quotation marks.

- Variables used to represent parameters in Lingo code and appear in *italics*; for example, *whichCastMember* is commonly used to indicate where you insert the name of a cast member in Lingo code. The names of sample titles such as *Noh Tale to Tell* also appears in italics.

- Paragraphs next to the ***Tip>*** note contain useful information that can help with using Lingo but aren’t essential for learning Lingo.

- The continuation symbol (¬) (which you enter by pressing Option-Return or Option-L) indicates that long lines of Lingo have been broken onto two or more lines. Lines of Lingo that are broken up this way are not separate lines of code. When you see the continuation symbol in this manual, you can recombine the lines when you type them into a script window.

Learning Lingo with online files

Director 4 includes three ways to learn Lingo in addition to this guide:

- The Lingo Expo
- The Learning Lingo tutorial movies
- The online help and help settings files.

Each of these is discussed in the following sections.

The Lingo Expo

A first step for learning Lingo is to explore the sample movies in the Lingo Expo folder. You might already be familiar with the movies from studying the score and cast, but look at them again and notice what Lingo achieves.

To illustrate typical ways that these features are used, your Director package contains three sample titles:

- *Noh Tale to Tell,* based on a story from traditional Japanese theater, uses Lingo to let users travel back and forth through parts of the story.
- *Furniture + Philanthropy,* an information kiosk about an exhibition of furniture design, demonstrates a wide range of user interactivity and information handling.
- *MECH*, a source of simulated gears, balls, and ramps that you can drag onto the stage and combine to create complex machines, uses advanced Lingo.

These samples are common types of movies that people create with Director. They are intended to give you the background you need to create a similar project yourself.

The *Navigator* provides a convenient way to play and study the sample titles. You can use it to play any of the titles and study the Lingo that each title contains. In addition, the *Navigator's* Lingo index shows you examples of Lingo used to implement specific features in the titles.

Remember that the Lingo in these movies increases in complexity. *Noh Tale to Tell* contains basic Lingo. *Furniture + Philanthropy* uses more complicated Lingo, but you can learn very useful Lingo tricks from it with a reasonable amount of effort. *MECH* uses advanced Lingo. However, you do not need to understand all of Lingo before you can use the parts of it that are demonstrated in these movies.

Explanations in this manual often refer to Lingo used in the sample movies. This helps you better understand the feature by seeing it in the context of a movie. But besides teaching you Lingo, these movies show you typical Lingo scripts that you can imitate in your own movies, and illustrate approaches to structuring Lingo for an entire movie.

Tutorial movies

The Director package also includes tutorials that contain the building blocks for the sample movies. The step-by-step tutorials throughout this book use the following movies in the Learning Lingo folder:

- Storybook (folder). This folder contains tutorial movies that are excerpted from *Noh Tale to Tell*. The individual movies are:
  - "BasicNav," in which you write Lingo that determines how the movie can branch to different frames or let the user decide when to pause or continue
  - "BNOM," which shows you how to write Lingo that returns to the frame that the movie was at when the movie branched
  - "IfThen," in which you write Lingo that chooses an action based on conditions and events in the movie
  - "ITOM," a companion movie to "IfThen," which "IfThen" can branch to
  - "MyMenus," which shows you ways to install custom menus
  - "Shared.dir," which contains the shared cast used by the tutorial movies in the Storybook folder.
  - "Sound," in which you control sound levels and fade sound in and out
  - "Timeout," which has you specify what happens when no one uses the keyboard or mouse for a specified period of time.
- Kiosk (folder). This folder contains tutorial movies that are excerpted from *Furniture + Philanthropy*. The individual movies are:
  - "Cursors," in which you write Lingo that changes the cursor to indicate movie conditions
  - "Rollover," in which you write Lingo that checks when the cursor is over specified sprites
  - "UserKeys," an exercise that teaches you how to combine display text and checks which keys the user presses.

- Simulation (folder). This folder contains several movies that are simple examples of the type of Lingo used in *MECH*. The individual movies are:
  - "INTERACT," a movie used by the tutorial movie "MIAW," is also in this folder
  - "Lists," which lets you create a simple list
  - "MIAW," an exercise in which you create a movie that can create a window and play a separate movie in it
  - "PareDone," which is a finished version of the movie you create in "Parents"
  - "Parents," an exercise that has you write Lingo that creates more than one child object from one parent script
  - "SimpDone," which is a finished version of the movie you create in Simple
  - "Simple," an exercise that has you write a simple parent script
  - "Wallcovering movie," a QuickTime movie used by the movie "MIAW."

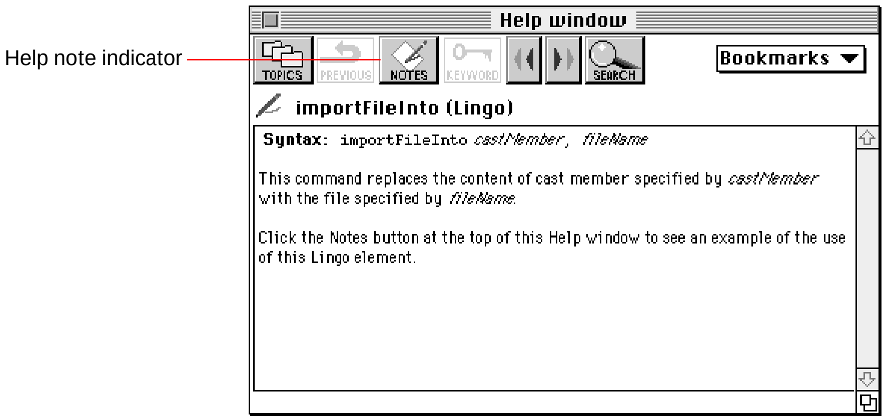
Not all the Lingo in the sample movies is explained in the tutorials. The best way to learn how the movies were put together is to take them apart and examine their scripts individually.

Online help examples

The online help system for Director includes a description of each Lingo command. In addition, it includes an example of each command in use that you can copy and paste into a script window. Both the Director 4.0 help and the help settings file must be in the same folder as your application.

To use the help settings file:

1. Launch the Director application.
2. Open a script window.
   The Lingo menu appears.
3. Choose Help Pointer from the Apple menu or press Command-?.
   Your cursor turns into the help cursor.
4. Choose the command you would like help on from the Lingo menu.
   The help screen for that command appears.



Syntax: importFileInto *castMember, fileName*

This command replaces the content of cast member specified by *castMember* with the file specified by *fileName*.

Click the Notes button at the top of this Help window to see an example of the use of this Lingo element.

5. Click the help note indicator.
   The help note screen appears.
6. Copy any script fragments you would like to use.
7. Click OK to cancel the help note dialog box.
8. Paste into the script window.

Note *The Lingo that you copy from the help window is an example, and not necessarily in the final form that it needs to be to run in your movie. Expect to change references to items such as cast members, variables, and markers to the appropriate versions before the Lingo runs in your movie.*

New Lingo features in Director 4.0

Director 4.0 introduces the following new Lingo features:

- Windows that can play auxiliary Director movies with each movie's Lingo in effect. You can create and delete these windows as your primary movie plays. (See Chapter 9, "Movies in a Window.")
- Parent scripts, which let you create a set of similar objects on command from the same parent script. This is a simpler alternative to factories in earlier versions of Director. (See Chapter 10, "Parent Scripts and Child Objects.")
- Removing Lingo source code text from final movies. This lets you protect a movie's scripts when you create projectors and protected movies, and immediately check whether a script's syntax is valid before playing the movie.
- An easier way to enter, edit, and check scripts. Movie, frame, and sprite scripts are now treated as cast members. Director 4.0 also checks whether a script's syntax is correct as soon as you enter the script, which saves rewinding and playing the movie to test a script's syntax. (See Chapter 1, "Script Basics.")

  An important difference is that when you edit an existing script and then press Enter, your changes are made to the original script. This differs from earlier versions of Director, in which pressing Enter after editing an existing script automatically created a revised version of the script and assigned it a new script number.
- An improved, more consistent approach to handling messages and defining event handlers. (See Chapter 3, "Concepts.")
- Additional cast and file management elements. The new elements let you use Lingo to do such things as find out or change cast member's type, color depth, palettes, and other attributes; copy and paste cast members to and from the Clipboard; and move cast members within the cast window.

- Lists, which offer a simpler way to hold a series of values. This is an alternative to using factory arrays with `mGet` and `mPut`. New Lingo elements for manipulating values are also included.
- New math features, including trigonometric and exponential functions.
- Ability to save modified files using Lingo.

For a complete list of the Lingo elements that are new in Director 4.0, see Appendix A "Lingo Changes" in the *Lingo Dictionary*.

Using outdated Lingo

Director 4.0 introduces some changes in Lingo syntax. As a result, Director 4.0 does not support all Lingo in movies created with earlier versions of Director. Director 4.0 automatically updates some outdated syntax when opening an old movie. Also, when the Allow Outdated Lingo checkbox in Director's Movie Info dialog box is turned on, Director does run some types of outdated Lingo. However, you cannot use any outdated Lingo when you create new scripts in Director 4.0.

The following table lists the Lingo syntax that is outdated in Director 4.0 and indicates the effect of turning on Allow Outdated Lingo:

Feature in earlier version	Changes in Director 4.0	Updated when movie opens?	Can be used with Allow Outdated Lingo?
Sprite scripts	Sprite scripts now require that `on mouseUp` and `on mouseDown` events be declared explicitly.	No	Yes
Frame scripts	Frame scripts now require that `on enterFrame` and `on exitFrame` be declared explicitly.	No	Yes
Text window scripts	Scripts in text cast members are now in movie scripts.	Yes	Not applicable, Director 4.0 automatically updates
Macro syntax	Macro syntax no longer used. Use `on ...` to define handlers.	Yes	Not applicable, Director 4.0 automatically updates
Loose syntax at end of line	No longer ignores extra text at the end of a line.	No	Yes
Handlers that contain improper syntax	Now generate a syntax error.	No	Yes
Using variables before they are defined	Now generates an error when the script is compiled.	No	Yes
Octal syntax (e.g., `A11`) for cast names	Octal syntax is no longer supported. Octal terms are treated as variable names.	No	Yes
`on stepMovie`	Replaced by `on enterFrame`.	No	Yes

Chapter 1

Script Basics

This chapter introduces basic concepts about scripts.

The first section of this chapter, "Introduction to scripting," describes what scripts are and how simple scripts behave. The second section, "Debugging," describes the process used to identify and correct errors in scripts. The third section, "The scripting process," describes what writing scripts is like and the type of effort you should expect to make. Read these sections if scripting is completely new to you.

The final section, "Entering scripts," describes how to use Director's score and script windows to enter, edit, and assign scripts in the score and script windows. The method for doing this is new in Director 4.0. Even if you are familiar with Lingo, you should read this section to see how this part of the interface has changed.

Introduction to scripting

The basics of using scripts are similar in any scripting language such as Lingo. This section introduces those basics and guides you through a few simple exercises that show the basics in use.

What a script is

Scripts are combinations of words that convey information and instructions. In this way, they are similar to the language you speak every day. Scripts vary in complexity.They can consist of a single one-word statement or of many statements similar to paragraphs.

Lingo and other scripting languages have certain elements that you use and rules that you follow to create statements. You use the statements in much the same way as when you speak.

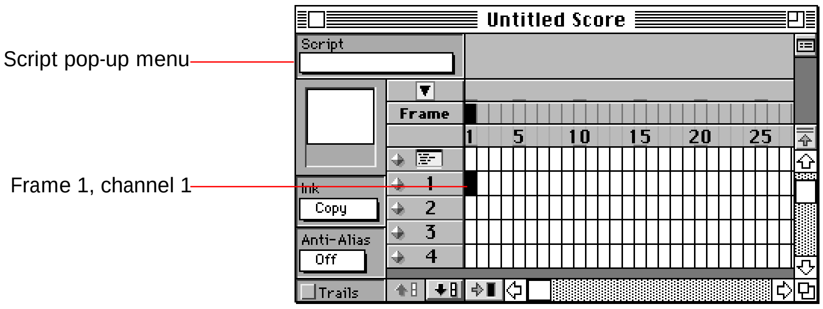
To see a simple Lingo statement, complete the following simple scripting exercise. The exercise is intentionally brief and meant to show you basic Lingo concepts. Later information shows you advanced uses for this and related Lingo.

Before you start:

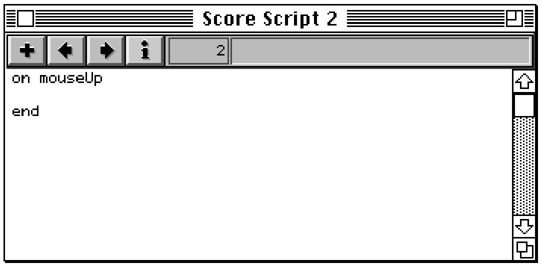
1. Start the Director application.
2. Choose New from the File menu.
3. Turn looping on in the control panel.

To create a simple script:

1. Open the score and select channel 1 in frame 1 of the score window.



2. Open the tools window by choosing Tools from the Window menu.
3. Click the button tool to select it in the tools window; then drag it on the stage to create a button.
   Select any button color and text setting you want.
4. Type the word Beep in the button's text field.
5. With the button still selected, choose New from the Script pop-up menu in the upper left corner of the score window.
   The score script window, in which you write scripts, appears.



```
on mouseUp

end
```

The score script window already contains the lines `on mouseUp`, followed by a line with a blinking cursor, and a line with the word `end`.

6. **Type beep in the script window.**
   The word appears at the blinking cursor. It is important that you type scripts correctly. Lingo can't interpret misspelled words or incorrect punctuation.

7. **Press Enter or click the close box in the upper left corner of the script window.**
   Always press Enter or click the close box to enter a script. You might expect to press Return, but Return starts a new line in a script, the same as in a word processor application.

To test what you made:

1. **Rewind and play the movie.**
   The movie loops in frame 1 because you turned looping on.

2. **Click the button.**
   You should hear a beep. If you don't hear a beep, try the steps that you followed to create a simple script again. Make sure you typed `beep` correctly and pressed Enter or clicked the close box when you were done in the script window.

To see a simple example of how you can combine Lingo elements:

1. Create a second button in channel 2 of frame 1 of the score you have been working with.
2. Type `3 Beeps` in the button text field.
3. With the new button still selected, choose New from the Script pop-up menu.
   A new script window appears.
4. Type `beep 3`; then press Enter or click the close box.
5. Play the movie and click the buttons.
   The button named "3 Beeps" has the computer beep three times when you click it.
6. Stop the movie you just made by pressing the stop button on the control panel or pressing Command-period.
7. Press Command-S to save the movie you have been creating.
   The first time you save the movie, a directory dialog box appears.
8. Type the movie's title in the Save Movie As field.
   Give the movie any name you want.
9. Click Save.

As you might expect, adding the number `3` after the term `beep` created a script that has the computer beep three times when the script runs. Many Lingo commands allow you to add parameters that further specify what the command does.

You just wrote scripts attached to sprites. When you clicked a sprite while the movie was running, Director checked whether the sprite had a script attached. The movie responded by running the script. In this case, the script was either `beep` or `beep 3`.

To write a different type of script:

1. Open the movie you just made if the movie is not currently open.
2. Turn off looping in the control panel.
3. Select frames 1 to 5 in channel 1 and channel 2; then use in-between to copy the Beep and 3 Beeps buttons into each frame.
4. Select frame 5 of the script channel.
5. Create a new script window by choosing New from the Script pop-up menu in the score.
   A new script window appears. When you create a new script window for a script in the script channel, the window already contains the line `on exitFrame`, followed by a line with a blinking cursor, and then a line with the word `end`.
6. Type `go to frame 1` in the script window, and then press Enter.
   The script you entered becomes attached to the script channel for frame 5.
7. Rewind and play the movie.
   By looking at the frame number display in the lower left corner of the control panel, you can see that the movie loops from frame 5 back to frame 1, even though you turned off looping in the control panel. The Beep and Beep 3 buttons you created still work the way their scripts instruct them to.
8. Stop the movie; then press Command-S to save what you have done.

You just wrote a script attached to the script channel. Scripts in this channel define what happens each time the playback head enters, exits, or is in the frame that the script is attached to.

The script `go to frame 1` instructs the movie to go to frame 1 each time the script is encountered. This loops the movie from frame 5 back to frame 1.

As the movie plays, the scripts assigned to the buttons have the same effect as before. Clicking a button makes the computer beep.

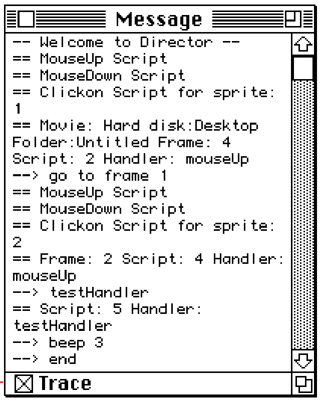
Messages

Director controls a movie by sending and receiving messages among the movie's objects—items such as cast members, sprites, and scripts. Messages can be generated by a script, user action with the keyboard or mouse, or something that occurs in the system.

When events— such as clicking the mouse or exiting a frame—occur, Director sends a message describing the event to a series of objects. The terms `mouseUp` and `exitFrame` are examples of such messages. When an object has a script that is set to respond to the particular message, the instructions in the script are carried out.

To see how messages are passed as a movie plays:

1. Choose Message from the Window menu.
   The message window appears.



```
-- Welcome to Director --
== MouseUp Script
== MouseDown Script
== Clickon Script for sprite:
1
== Movie: Hard disk:Desktop
Folder:Untitled Frame: 4
Script: 2 Handler: mouseUp
--> go to frame 1
== MouseUp Script
== MouseDown Script
== Clickon Script for sprite:
2
== Frame: 2 Script: 4 Handler:
mouseUp
--> testHandler
== Script: 5 Handler:
testHandler
--> beep 3
--> end
```

2. Turn on the Trace checkbox in the lower left corner of the message window.
   When Trace is on, the message window displays messages and scripts that are encountered as the movie plays.

3. Play the movie in which you created the Beep and 3 Beeps buttons.

4. Click the Beep and 3 Beeps buttons as the movie plays.
   Notice that when you click the button in channel 1, the message window displays the message `Clickon Script for sprite: 1`. When you click the button in channel 2, the message window displays the message `Clickon Script for sprite: 2`. The message window also displays the script attached to the sprite.

You just used the message window to trace a message that was sent when you clicked the mouse. You also saw that the message window displays lines of scripts as they are executed.

Handlers

A handler is a set of Lingo statements attached to an object. The statements are preceded by the word `on` and the name of the message that the handler should respond to.

The following is what occurred in the scripts you wrote in "What a script is," earlier in this chapter. The first line `on mouseUp` was actually the first line of a handler. The line `on mouseUp` had the statements in the handler run whenever the mouse button was released after you clicked the object that had the script attached. A `mouseUp` is one of Lingo's predefined events that can call handlers. As you'll see in the next exercise, you can also define handler names yourself.

By attaching the `on mouseUp` handler to a sprite, you tell Director to execute the handler only when the mouse button is released after clicking the sprite that has the handler. When you click other places on the stage, the handler doesn't execute.

When an object receives a message that calls or refers to the handler, the instructions in the handler are executed in the order the statements appear.

The following exercise shows how to create a simple handler. In this case, the name of the handler is one you define.

First write the script for the handler:

1. Open the movie that contains the Beep and Beep 3 buttons you just created if it isn't currently open.
2. Choose Script from the Window menu.
   A script window appears.
3. Click the add sign in the upper left corner of the script window.
   A new script window appears.
4. Type the following:
   ```
   on testHandler
     beep 4
   end
   ```
5. Press Enter or click the close box to enter the script.

Handlers always begin with the word `on` followed by the handler's name and end with the word `end`. Lines of Lingo within the first and last lines of the handler run when the handler is called by a message. Handlers, such as `testHandler`, that have names you define yourself require a line of Lingo that calls the handler at the proper time. This is different than handlers such as `on mouseUp`, whose names are predefined in Director. Predefined handler names in Director are described in the section "Describing events," in Chapter 3.

To give objects in the movie a way to call the script in the handler, put the handler's name in a script attached to the object.

1. Select frame 1 in channel 3 of the movie.
2. Use the button tool in the tools window to create a button cast member.
3. Type `Handler` in the text field for the button.
4. Select frames 1 through 5 and in-between the button sprite across all five frames.
5. With frames 1 through 5 still selected, choose New from the script pop-up menu.
   The script window appears.
6. Type `testHandler` in the script window; then press Enter to enter the script.
   `testHandler` is the name of the handler you wrote in the movie script. The script is entered for all sprites that are selected when you press Enter.

To test what you created:

1. Open the message window and turn on the Trace checkbox.
2. Rewind and play the movie.
3. Click the handler button.
   As you expect, Lingo in `testHandler` has the computer beep four times.

You just wrote a handler attached to a sprite. You can follow what happens by checking the message window. When the sprite is clicked, the `clickOn` message activates the `testHandler` statement attached to the button. Director finds the contents of this script in `testHandler`, which you defined in a movie script.

Debugging

Very few scripts do what you want the first time they run. Often the script has an error in its syntax: possibly a word is misspelled or a small part of the script is missing. Other times the script might work, but doesn't produce the expected result.

Getting the script to run correctly requires identifying the problem and then correcting it. This process is commonly called debugging.

Debugging scripts is an important part of scripting a movie. The best ways to determine what the problem is in a script vary depending on the type of script and the Lingo involved. The explanations of specific Lingo features later in this manual include debugging techniques that are useful for the specific feature being discussed. For additional debugging methods, see *Tips & Tricks*.

Some simple debugging tricks are useful for any script. When a script fails, make sure that:

- Terms are spelled correctly, spaces are in the correct place, and necessary punctuation is used. Lingo can't interpret incorrect syntax.
- All necessary parameters of the Lingo statement are present. Many Lingo structures expect certain terms to be present. The specific parameters depend on the individual element. See the *Lingo Dictionary* or online help to determine any additional parameters required by an element.
- Quotation marks surround the names of cast members, labels, and strings of text used within the statement. See the section, "Expressing literal values," in Chapter 3 for more information about which items require quotation marks.
- Values for parameters are correct. For example, using an incorrect value for the number of beeps that you want the `beep` command to generate would obviously give you the wrong number of beeps.

- Values that change—such as variables and the content of text cast members—have the new values you want. You can check this by using the `put` command as described later in this section. Variables are described in Chapter 3, "Concepts." You do not need to understand them now.

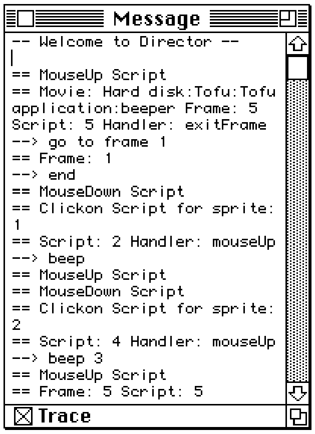
It is good practice to break your Lingo into smaller sets of statements and test each one as you write it. This keeps potential problems isolated to areas that are more easily identified.

The first debugging test occurs when you close the script window. Lingo gives you an error message if the script contains incorrect syntax when you close the script window. The message usually includes the statement in which the problem was first detected.

When the script window closes without an error message, the script might still contains a bug. For example, the names of cast members, numbers of sprites, or parameters you include could be incorrect values.

If the problem isn't a simple one, you can often find which script the problem is in by using the message window to follow how scripts are being executed as the movie runs.

To see how the message window displays information about scripts in the movie, play the movie that you made in the previous section with the Trace checkbox in the message window turned on.



```
-- Welcome to Director --

== MouseUp Script
== Movie: Hard disk:Tofu:Tofu
application:beeper Frame: 5
Script: 5 Handler: exitFrame
--> go to frame 1
== Frame: 1
--> end
== MouseDown Script
== Clickon Script for sprite:
1
== Script: 2 Handler: mouseUp
--> beep
== MouseUp Script
== MouseDown Script
== Clickon Script for sprite:
2
== Script: 4 Handler: mouseUp
--> beep 3
== MouseUp Script
== Frame: 5 Script: 5
```

When a script is activated, the message window displays each line of the script as it is executed. An arrow made up of a double hyphen and right angle pointer (`-->`), precedes each script line. You can often locate the problem by following the display in the message window to isolate where the problem occurs.

You can also use the `put` command to display messages about the script's progress as it runs.

To see how the `put` command works:

1. Open the movie script for the movie in which you created the handler named testHandler.
2. Add the line `put "the handler is done"` before the last line of the handler.
   When you have finished making this change, the last two lines of the handler should be

   ```
     put "the handler is done"
   end
   ```
3. Turn on the Trace checkbox in the message window.
4. Play the movie and click the handler button.
   Notice that when the movie reaches the line `put "the handler is done"`, the message window displays the message "`the handler is done`".
5. Turn off the Trace checkbox in the message window.
6. Replay the movie and click the handler button again.
   Notice that the message "`the handler is done`" still appears in the message window even though the Trace checkbox is not on.
7. Close the movie without saving changes.

This is an example of how you can use the `put` command to follow a script's progress. The `put` command displays the specified result in the message window whether or not the Trace checkbox is on.

You can also use the `put` command to check the value of a variable, the content of a text string, or similar values by displaying them in the message window. More uses for the `put` command as a debugging tool are discussed with other Lingo features later in this guide.

Testing a statement in the message window

To use the message window to test a one-line Lingo statement, type the statement directly in the message window and press Return. Lingo executes the statement immediately.

If the statement is valid, the results are visible on the stage or in the message window itself if you have used the `put` command, which displays the result of the statement in the message window. If the script is invalid, an alert appears.

You can test a handler by writing it in a movie script or score script window and then calling it from the message window by typing the handler name in the message window and then pressing Return.

The scripting process

The scripts you've written so far are simple. You followed instructions to write short scripts for specific objects.

When you apply scripts to an entire movie, the quantity and variety of scripts increase tremendously. Deciding which Lingo to use, effective ways to structure scripts, and where scripts should be placed requires careful planning and testing as the complexity of the movie grows.

The most important part of scripting is formulating your goal and understanding what you want to achieve before you begin writing scripts. This is as important and typically as time-consuming as developing storyboards for your work.

Once you have an overall plan for the movie, you are ready to start writing and testing scripts. Expect this to take time. Getting scripts to work the way you want often takes more than one cycle of writing, testing, and debugging.

As the scripts you write in a movie become more complicated, it is easy to forget what the scripts are intended to do or what certain values in the script are for. To let you make useful notes about the script, Lingo lets you add comments—descriptive lines within a script that don't run as part of the script. Comments lines start with a double dash (`--`). Anything you type after the double hyphen and before Return doesn't affect the script.

To write a sample comment:

1. Open the movie script in which you created the handler named testHandler.
2. Before the first line of the handler, type two hyphens followed by the comment `this is a sample handler`.
3. Play the movie and press the handler button.
   Notice that the script executes the same as before and that the comment you added doesn't affect the script.

Scripts used later in this book include examples of comments. You can see how useful they are as you continue to work with scripts.

▶ **Tip** *You can comment or uncomment statements that are already written by selecting the statements and choosing Comment or Uncomment from the Text menu.*

Writing scripts

Several Director tools are used to write scripts:

◆ Script windows let you write, edit, and enter scripts. There are three types of script windows: movie script windows, score script windows, and script of cast member windows.

◆ The score lets you select frames and sprites that you can assign scripts to. When one or more cells is selected, choosing a script from the Script pop-up menu assigns the score script to the selected items. Scripts assigned in the script channel are assigned to the entire frame they occur in and are called frame scripts; scripts assigned to sprites are called sprite scripts.

  Choosing New from the Script pop-up menu opens a new score script window. Choosing an entry for an existing script and then clicking the script preview button to the right of the script pop-up menu opens the script window for that script. Any script that you enter or edit in the script window is assigned to the selected frames or sprites.

◆ The script button in the cast window or the Script option in the Cast Member Info dialog box for the selected cast members opens a script window for the cast member. Any script that you enter in this window is assigned to the cast member. (Cast members that have been assigned scripts display a small "L" in their thumbnails in the cast window.) A script assigned to a cast member is also known as a script of cast member.

◆ The cast window lets you open an existing movie script or score script by double-clicking the script cast member. You can copy a movie script or score script by selecting the script and choosing the Duplicate Cast Member command from the Cast menu.

◆ The Lingo menu lists Lingo elements that you can choose and insert into a script you are writing. After the element is inserted, the entry prompts you for any additional parameters that the element uses. You can use this menu to help remember what Lingo elements are available or an element's correct syntax.

When using these tools, you see that scripts can be assigned to different places in your movie. The advantages of different places that you might put scripts are discussed in the section "Strategies for placing handlers" in Chapter 3. For now, you just need to learn the mechanics of entering scripts.

Writing different types of scripts

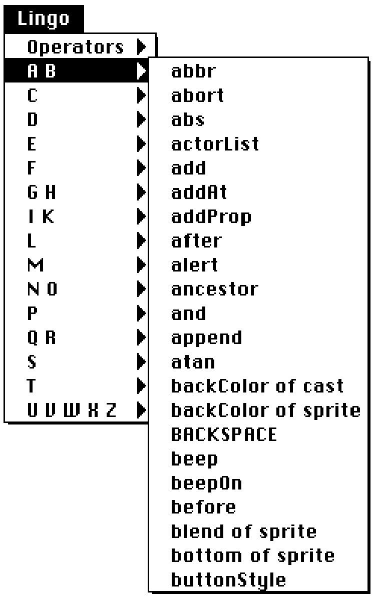
How you write and edit score scripts, movie scripts, and scripts of cast members depends on whether they are new or already exist:

- Open new score scripts by clicking the add (+) button in the upper left corner of the script window; or choosing New from the Script pop-up menu in the score and then clicking the script button in the upper right corner of the score.
- Open existing score scripts by choosing the script's number from the Script pop-up menu or double-clicking the script in the cast window.
- Open new movie scripts by clicking the add (+) button in the upper left corner of the script window.
- Open existing movie scripts by double-clicking the script in the cast window.
- Open scripts of cast members by selecting the cast member and then clicking the script button in the cast window or by clicking the Script button in the cast member's Cast Member Info dialog box.

Using the Lingo menu

You can use the Lingo menu to insert Lingo elements in any script window. When an element requires additional parameters, Lingo includes placeholder names that indicate the additional required information. When more than one argument or parameter is required, Lingo highlights the first one for you, so all you have to do is type to replace it. You have to select and change the other parameters yourself.

The Lingo menu is an alphabetical list of Lingo elements. Choosing an element pastes the item where the blinking cursor is in the message window or script window.



If you forget the syntax for a particular Lingo element, there's a quick way to get online help for Lingo words:

1. Press Command-? or choose Help Pointer from the Apple menu.
   The pointer changes to the Help pointer (a question mark).
2. Choose an item from the Lingo menu.
   The Help window opens to the description of that Lingo command.

Editing text for scripts

Entering and editing text in a script window is similar to entering text in any other text field:

- Select a whole word by double-clicking the word.
- Select a whole script by triple-clicking in the script.
- Use Undo, Cut, Copy, Paste, Clear and Select All from the Edit menu while you are in the script window.
- Enter statements one line at a time. The editor automatically formats the statements with indented lines when you press the Tab or Return keys.
- Use Option-Return to break up long lines of code. This allows Lingo to interpret the statement as one line, even though you wrap the text so that it can be read easily. This type of line break is similar to a soft return instead of a hard return.
- Press Enter to enter the script and close the script window when you are done. If the script syntax is not valid, an error message appears when you close the script window.

▶ ***Tip*** *Pay attention to how Lingo indents lines as you type. Because Lingo automatically indents statements when the syntax is correct, you can sometimes tell that there is a bug in a line if it doesn't indent properly.*

As you write new score scripts, they are assigned numbers. These numbers appear in the Script pop-up menu in the score. The number of the script also appears in its corresponding cell in the script channel. Numbers of movie scripts and scripts of cast members do not appear in the Script pop-up menu.

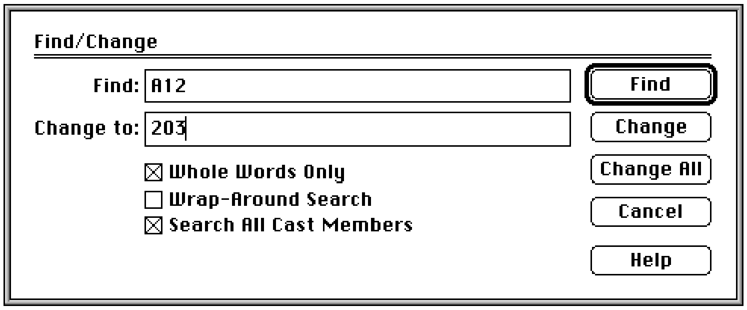
You can assign an existing script to any new location in the score simply by selecting the cells and then selecting the desired script from the Script pop-up menu.

Finding and changing text in scripts

The bottom half of the Text menu contains commands that are useful for finding and changing text in scripts, finding handlers, and commenting and uncommenting parts of scripts.

Use the Find/Change, Find Again, Change Again, Find Selection, and Find Handler commands to search through your scripts for a particular word or words.

▶ Choose the Find/Change command to open the Find/Change dialog box.


Find/Change

Find: A12

Change to: 203

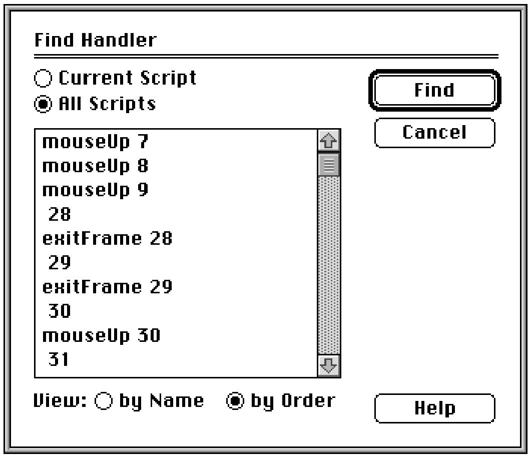
☒ Whole Words Only
☐ Wrap-Around Search
☒ Search All Cast Members

Enter the string you want to find and press Return or click the Find button. Find is not case-sensitive: ThisHandler, thisHandler, and THISHANDLER are all the same for search purposes.

Select the Whole Words Only checkbox to search only for whole words and not fragments of other words that match the word you are searching for. Select the Wrap-Around Search checkbox to have the search start over from the beginning after the search reaches the end. Select the Search All Cast Members checkbox to search all cast members of the same type; Director searches text or script cast members only, depending on your current selection.

If you want to change the string, enter the new string in the Change To field. Click Find to find the next occurrence of the string. Click Change to find the next occurrence of the string and change it. Click Change All to find all occurrences of the string and change them.

- ▶ Choose Find Again from the Text menu to search for the next occurrence of the string that you entered in the Find/ Change dialog box.
- ▶ Choose Change Again from the Text menu to change the string to the string you entered in the Change To field of the Find/Change dialog box.
- ▶ Choose Find Selection from the Text menu to find a string that is the same as a string you have selected in a script.
- ▶ Choose Find Handler from the Text menu to open the Find Handler dialog box.


Find Handler

[ ] Current Script
[✓ All Scripts]

```
mouseUp 7
mouseUp 8
mouseUp 9
 28
exitFrame 28
 29
exitFrame 29
 30
mouseUp 30
 31
```

View: by Name [✓ by Order]

The scrolling list in the Find Handler dialog box displays the names and script numbers of the movie's handlers. Find Handler is described in Chapter 6 of *Using Director,* "Menu Reference."

Removing a script from the score

You can remove a script from a cell in the score by selecting the cell and choosing 0 from the Script pop-up menu. Scripts in the pop-up menu that haven't been placed in at least one of the cells of the score are deleted when you close the movie, and the scripts are renumbered accordingly. Script number 0 is always available.

Script numbers

Script numbers for score scripts appear in the Script pop-up menu. When you create a new score script, the script is given the number of the first available slot in the cast window. A script is entered and assigned to the selected cells in the score whenever you press Enter or click the close box in the script window.

▶ ***Tip*** *When you make changes to an existing script, the changes are made to the original script. To use a script as a starting point for another script, copy the original script by selecting it in the cast window and choosing Duplicate Cast Member or copying and pasting the original script into a new slot in the cast window.*

Chapter 2

Working with Navigation

Navigation, giving users the ability to move around in and explore parts of a movie or branch to other movies, is the most common use of Lingo. This chapter shows you how to write navigation scripts in a way that you can easily duplicate in your own movies.

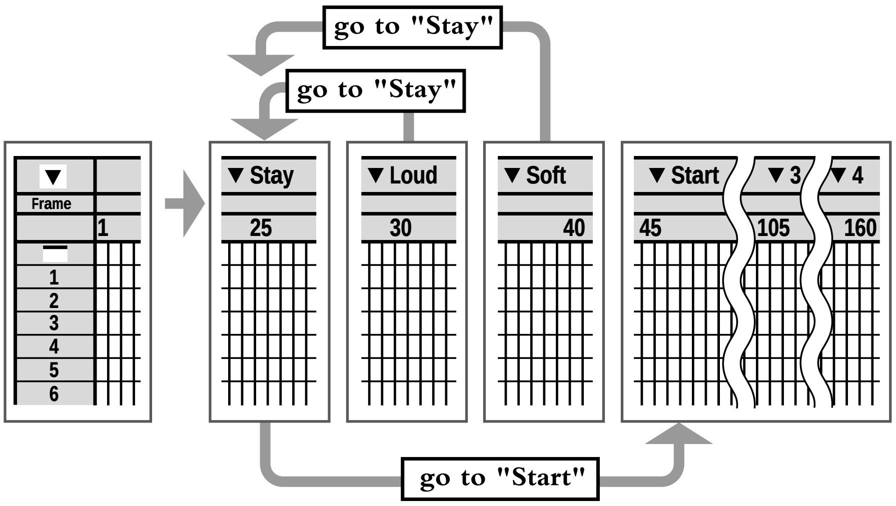
Writing navigation scripts is usually the part of Lingo that people learn first. If this is your first experience writing Lingo, you need to learn two things at once: the specific Lingo you are working with and some general concepts that always apply when using Lingo.

If you are a new Lingo user, you can follow the discussion of navigation scripts without too much background. You do need to read the section "Writing scripts" in Chapter 1 to see how to enter and edit scripts. After you master navigation scripts, see Chapter 3, "Concepts," for a better understanding of how Lingo sentences are structured, how to use Lingo syntax, and where you can place Lingo scripts.

Adding navigation

Including navigation in a movie gives users the ability to explore movies at their own pace, branch to parts of the movie that offer additional information, and pause or repeat the parts of a movie that interest them the most.

For example, the first section of *Noh Tale to Tell* can branch to one of several frames. The frame marked Stay displays the screen that you use to select a sound level; frames marked Loud and Soft contain Lingo that sets the sound level when you choose an item from the Volume menu; the frame marked Start is the beginning of the story. When you choose a sound level from the Volume menu or start the story, the playback head jumps to the appropriate frame as follows:



The Lingo you use to add navigation is relatively simple. Choosing where to place the possible paths for users is an important design consideration. The sample movies in the *Lingo Expo* are a good source of ideas about uses for navigation scripts.

The following sections describe Lingo scripts that implement navigation features in typical situations.

Creating loops

Looping a movie in a segment of one or more frames uses basic Lingo navigation scripts.

In Chapter 1, "Script Basics," you wrote a simple script that used the `go` command to jump to a different frame. By jumping to the beginning of a sequence of frames—or looping—you can create animation that appears to recycle. This section shows you how to use the `go` command and related Lingo to create a loop within a movie segment.

First, create the animation segment to use for the loop:

1. Choose New from the File menu.
2. Select frame 1 in channel 1 of the score.
3. Open the paint window.
4. Draw a small circle.
5. Drag the circle onto the stage using the place button.
6. Close the paint window.
7. With frame 1 in channel 1 of the score still selected, copy the cell.
8. Paste the sprite into frame 5 in channel 1.
9. Stretch the circle on the stage in frame 5.
10. In-between the circle from frame 1 to frame 5 in channel 1.

This sequence makes the circle appear to expand. You can make it appear to contract by reversing the sequence.

To reverse the sequence:

1. Copy the sprites in frames 1 through 4 into frames 6 through 9 in channel 1.
2. With frames 6 through 9 still selected, choose Reverse Sequence from the Score menu.
3. Play the movie.
   The circle appears to expand and then contract.

To have the frames loop using Lingo:

1. Select the script channel in frame 9.
2. Choose New from the Script pop-up menu at the top of the score.
   A new score script window appears. The lines `on exitFrame` and `end` already appear in the window.
3. Type `go to frame 1`; then press Enter.
   Pressing Enter enters the script and assigns it to the script channel for frame 9.
4. Open the message window and turn on the Trace checkbox.
5. Play the movie.
   The message window display and the playback head in the score show how the movie plays through the frames and then returns to frame 1.
6. When you are done, press Command-S to save your work. Name the movie whatever you like.

You just created an animation sequence and then had it loop by returning the playback head to the beginning of the sequence.You returned the playback head by using the script `go to frame 1`.

Many times you want a sequence to play through and then wait for the user to respond. For instance, when presenting a story such as *Noh Tale to Tell*, you probably want to wait at the end of each scene to let the reader finish reading before going on. A simple way to make the movie appear to wait is to have the segment loop in the last frame.

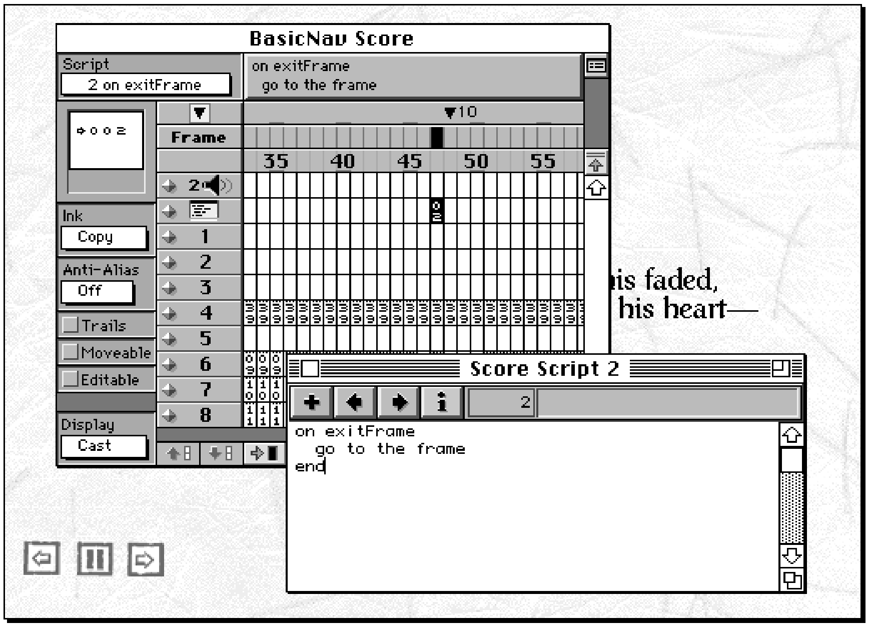
As an exercise, you can do this yourself by re-creating the Lingo that provides this feature in the movie *Noh Tale to Tell.*

First, play the unfinished version of the movie you will add the Lingo to:

1. Open the movie “BasicNav” in the Tutorials: Learning Lingo: Storybook folder.
2. Open the score and play the movie.
3. Notice that the movie plays from frame 47 (the end of scene 9) to frame 48 (the beginning of scene 10) without stopping.
   You can follow what happens by watching the playback head advance in the score.

To write Lingo that has the movie loop in the last frame of scene 9:

1. Select the script channel in frame 47.
2. Choose New from the Script pop-up menu.
   A new score script pop-up menu opens.



```
on exitFrame
  go to the frame
end
```

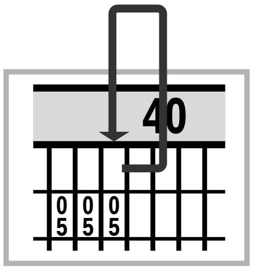
3. Type `go to the frame`, and then press Enter.
   The expression `the frame` is one way that Lingo refers to the current frame. Pressing Enter enters the script and attaches it to the script channel for frame 47.
4. Open the message window and turn on the Trace checkbox.
5. Play the movie.
   The message window display and the playback head show how the movie advances through frames and then loops in frame 47.

Navigation features that you use later in this chapter require that the same script be at the end of scenes 10 and 11a.

To add the same script to the end of scenes 10 and 11a:

1. Stop the movie if it is still playing.
2. Select the script channel in frame 148.
3. Choose the number of the script you just wrote from the Script pop-up menu.
   The script is now assigned to frame 148 also.
4. Select the script channel in frame 167.
5. Choose the number of the script you just wrote from the Script pop-up menu.
   The script is now assigned to frame 167 also.
6. When you are done, press Command-S to save your work.

The arrow in the following figure illustrates how the playback head loops by re-entering the frame. A movie set up this way continues to loop in the specified frame until it receives further instructions. The next section describes how you can also use the `go to` command and markers to provide ways for the user to branch to a different part of a movie.



Moving between sequences

Authors often want to let viewers jump to a movie sequence they choose. You can give users this choice through scripts that send the playback head to different locations in a movie.

This section first tells you ways to identify the frames you want to go to by using frame numbers, markers, and labels. It then shows you how to write scripts that let someone move back and forth in the parts of a story, or choose one of several places to go to. You add this Lingo to the movie “BasicNav,” for which you wrote a looping script in the section “Creating loops,” earlier in this chapter.

Identifying movie locations

From working with the score and basic Lingo, you have seen how frame numbers identify specific places in a movie. These identifiers are important in movies that can jump between segments.

As you saw when you used the Lingo statement `go to frame 1`, Lingo can refer to frames explicitly by frame number. For convenience, Lingo offers several ways to refer to the current frame and frames before or after it:

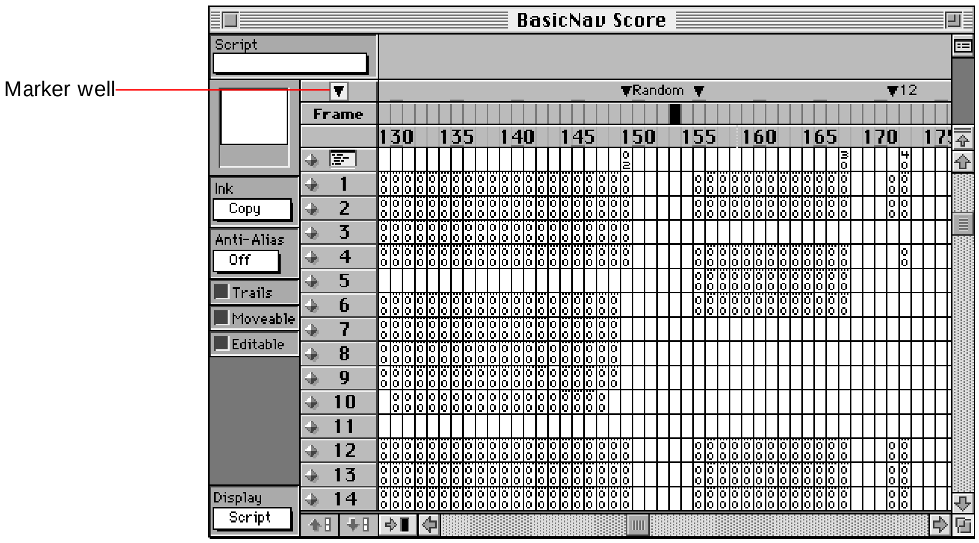
- To refer to the current frame, use the function `loop` or `the frame`. For example, `go loop` or `go to the frame` loops the movie in the current frame. The statement `put the frame` has the message window display the number of the current frame.
- To refer to a frame that is a specific number of frames before the current frame, use `the frame` followed by a minus sign and the number. For example, `go to the frame - 1` refers to the frame just before the current frame. The statement `put the frame - 2` has the message window display the number of the frame two frames before the current one.

- To refer to a frame that is a specific number of frames after the current frame, use the function `the frame +` followed by the number. For example, `go to the frame + 1` refers to the frame just after the current frame. The statement `put the frame + 2` has the message window display the number of the frame two frames after the current one.

As the size of your movie increases, it becomes more difficult to remember what's in different sequences of frames. References to frame numbers may also become incorrect if you rearrange, insert, or delete frames in the score. Markers identify places in the movie by the same name or whether the markers are before or after the current location. These references remain the same no matter how much you edit the score.

To see how to create a frame marker and label:

1. Open the movie "BasicNav" that you worked with in the section, "Creating loops," earlier in this chapter.
2. Open the score and scroll so that frame 155 is visible.
3. Click the marker well and then drag a marker to frame 155.
4. Release the mouse button in frame 155.
   A black marker appears at the top of frame 155, as shown in the following figure.



   The marker shows a blinking cursor, ready for you to enter a label for the frame.

5. Type `11a` and press Return.
   You labeled the marker 11a to indicate that this is the beginning of scene 11a.
6. Press Command-S to save your work.

Lingo can specify a frame with a marker by referring to its marker label. For example, the Lingo statement `go to "11a"` has the playback head jump to the frame that has the marker labeled 11a. In the "BasicNav" movie, the statement `go to "11a"` is equivalent to `go to frame 155`.

Lingo can also refer to markers by how many markers they are ahead or behind the current frame. For example, `marker(0)` refers to the current marker, `marker (1)` refers to the first marker after the current frame, `marker (2)` refers to the second marker after the current frame, and so on. Use a minus sign before the number for markers before the frame. For example, `marker (- 2)` refers to the second marker before the current marker.

Moving forward and backward

By including a `go to` statement in a script assigned to a cast member or sprite, you can let users jump to a different location in a movie by clicking the object that has the script assigned to it.

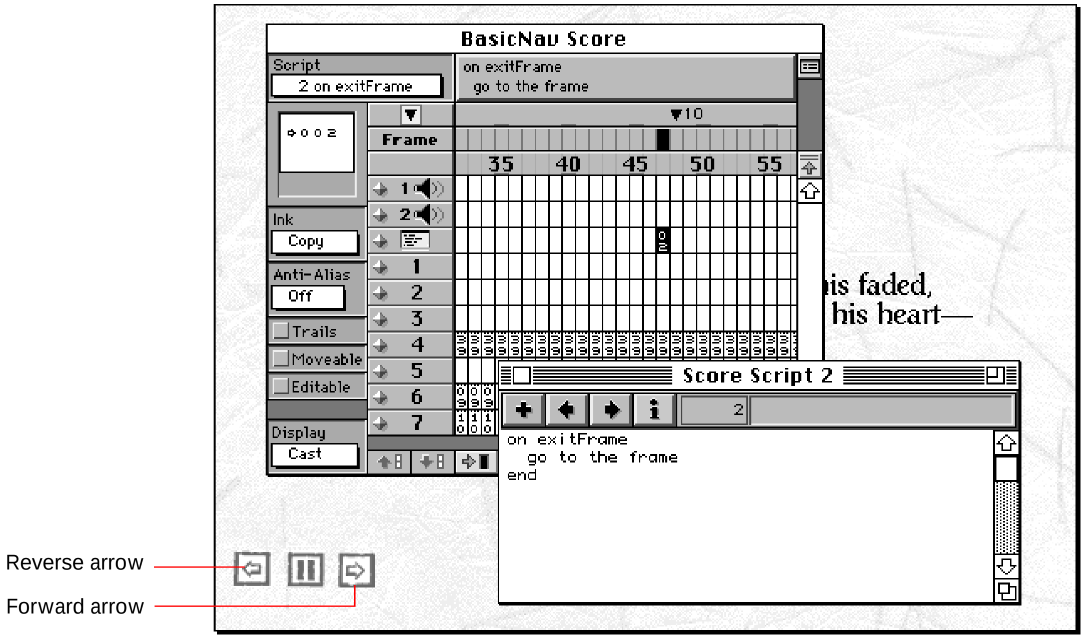
- When you want the object on the screen to always respond the same way when it is clicked, assign the script to the cast member. For example, write a script for a button that always goes to the same help section or the beginning of the movie by assigning the script directly to the button cast member. Otherwise, you would need to write the script for every sprite of the button that is in the score.
- When you want clicking the object to have different results in different frames, it is better to assign the script to the sprite. For example, a button that the user clicks to answer yes to a question could require a different response to different questions. Assigning scripts to sprites lets you control the response depending on the situation.

In this section, you re-create the Lingo that was used to go forward and backward in the movie *Noh Tale to Tell.*

Before you write the scripts, first see what the movie does without them:

1. Open the movie "BasicNav" for which you wrote a looping script in the section, "Creating loops," earlier in this chapter.
2. Open the score.
   The score duplicates segments for scenes 9 through 12 of *Noh Tale to Tell*.
3. Play the movie.

The movie stops at frame 47, which is where you placed the frame script that has the movie loop in frame 47 when you performed the exercise in the section, "Creating loops."

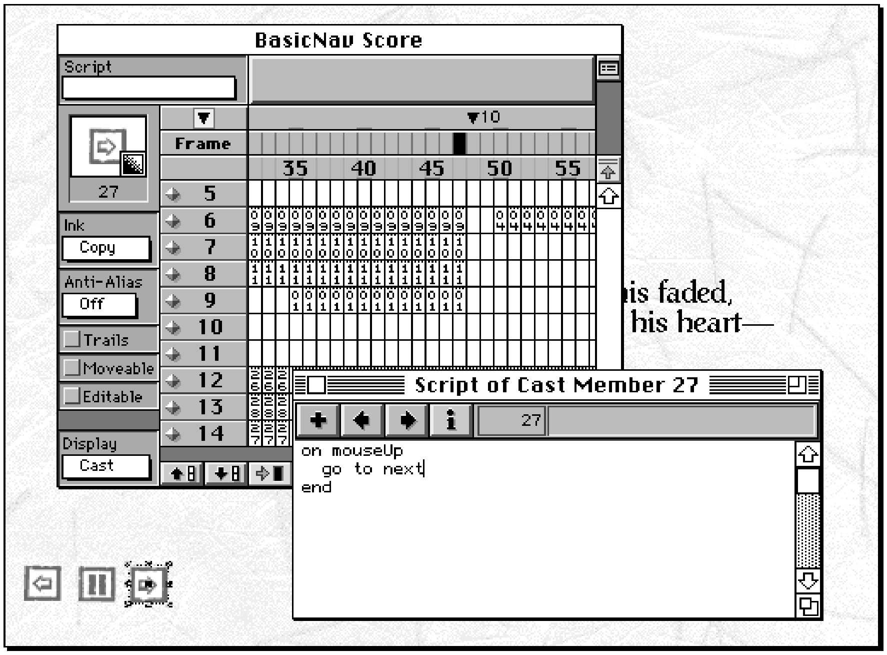


```
on exitFrame
  go to the frame
end
```

Now add Lingo that has the playback head jump to the next marker in the movie. In this exercise, you assign the script to the cast member used for the forward arrow in the movie's control panel. When the Lingo is present, clicking the arrow has the movie go on to the next scene.

To write the script that lets the user go to the next marker:

1. Open the cast window.
2. Select cast member 27, the forward arrow.
3. Click the script button at the top of the cast window.
   The script of cast member window appears.



```
on mouseUp
  go to next
end
```

Script of cast member windows automatically open with the lines `on mouseUp` and `end` already written for you.

4. Type `go to next`.
   When you are done the entire script should be:

   ```
   on mouseUp
     go to next
   end
   ```

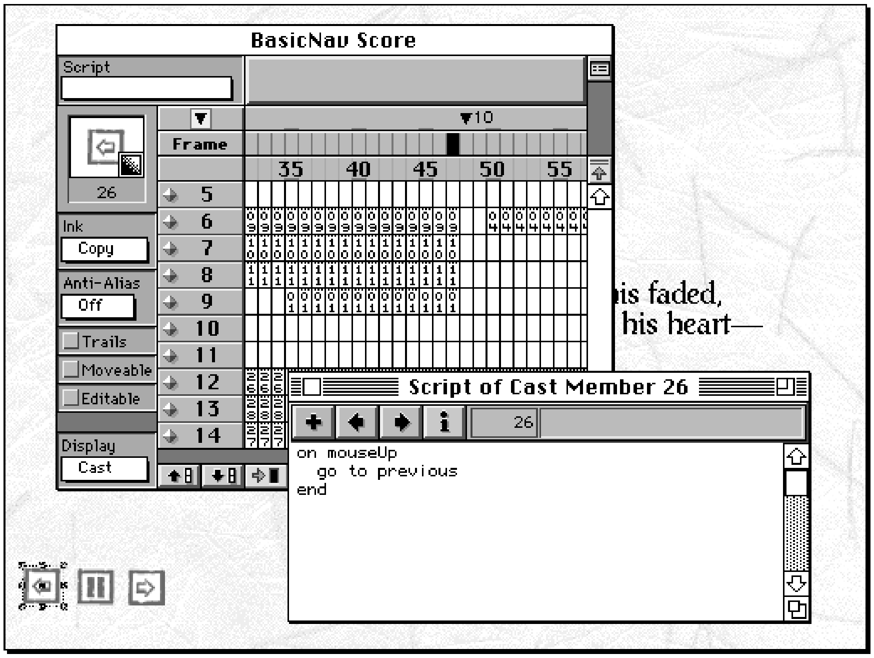
5. Press Enter or click the close box to enter the script.
   The script is now assigned to the forward arrow cast member. The script runs when someone clicks and releases the mouse button after clicking the forward arrow.

The script `go to next` sends the playback head to the next marker in the score. The result in this sample movie is that when you click the forward arrow, the movie goes on to the next scene. To have the movie loop in the current frame, use the statement `go to loop`. To have the movie back up to the beginning of the current scene, use the statement `go to previous`, which sends the playback head to the marker before the playback head. To have the movie back up to the beginning of the scene before the current scene, use the statement `go to marker(-2)`.

▶ ***Tip*** *When writing loops and go to commands, it is useful to leave the score open so that you can track where the playback head goes as the movie plays.*

To write Lingo that has the playback head go to the beginning of the current segment when the user clicks the reverse arrow in the control panel:

1. If it isn't open, open the movie "BasicNav" that you have been working with.
2. Select cast member 26, the reverse arrow.
3. Click the script button at the top of the cast window. The script of cast member window appears.



```
on mouseUp
  go to previous
end
```

4. Type `go to previous`.
   When you are done the entire script should be:

   ```
   on mouseUp
     go to previous
   end
   ```

5. Press Enter or click the close box to enter the script.
   The script is now assigned to the reverse arrow cast member. The script runs when someone releases the mouse button after clicking the reverse arrow.

To test the scripts you wrote for the forward and reverse arrow cast members:

1. Open the message window and turn on the Trace checkbox.
2. Rewind and play the movie.
3. Click the forward arrow after the movie reaches frame 47, which is the end of scene 9.
   The movie jumps to the next marker, labeled “Scene 10,” when you click the forward arrow.
4. Click the reverse arrow after the movie reaches the ends of scenes 9 and 10.
   The movie returns to the marker at the beginning of the current scene and plays the scene again.
5. Click the forward arrow again when the movie reaches the end of scene 10.
   The movie plays to the end.
6. Press Command-S to save your work.

The script `go to next` sends the playback head to the next marker in the score. The result in this movie is that the movie goes on to the next scene each time you click the forward arrow that you assigned the `go to next` script to.

Another common use of a script for a cast member is to create a button that sends the playback head to the same frame regardless of where the playback head is. For example, *Noh Tale to Tell* has a marker labelled "Start," which marks the beginning of scene 1. You can have clicking a button or other cast member always return the movie to the start of the movie by assigning the following script to the cast member:

```
on mouseUp
  go to "Start"
end
```

Lingo's `go to` scripts have many uses. You can get more ideas of how to use them by studying the sample movies in the *Lingo Expo*, which provides many examples of these scripts.

Pausing and continuing a movie

Many times you will want to let the user finish looking at a screen or make a decision before going on. The `pause` and `continue` commands provide these choices—allowing you to pause the movie until the user clicks a particular sprite.

To see how to use these commands, add a pause before the second block of text begins to appear in scene 10 of *Noh Tale to Tell*. Then add a `continue` command that has the movie continue when the user clicks the forward arrow on the control panel.

To add the pause:

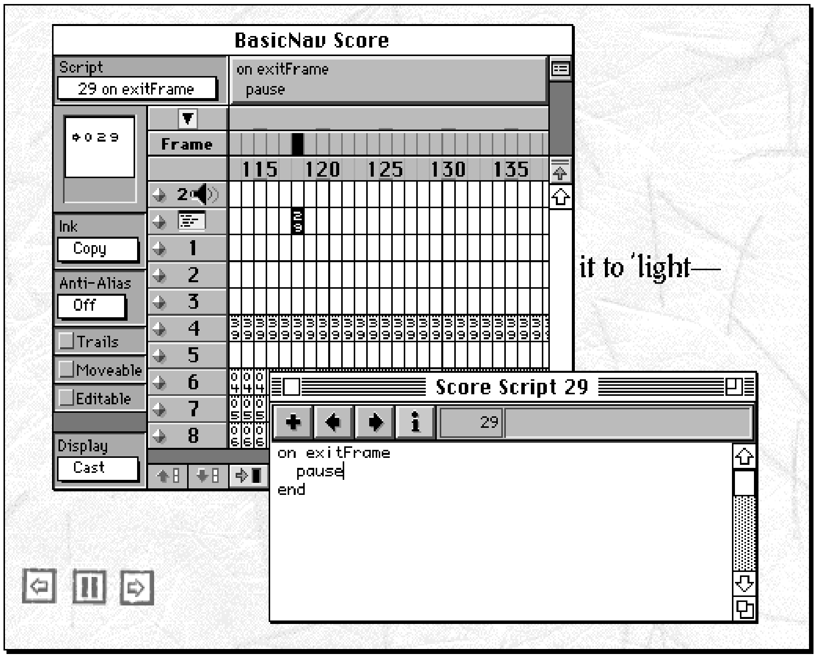
1. If it isn't open, open the movie "BasicNav."
2. Open the score.
3. Select the script channel for frame 118.
   At frame 118, the third block of text has appeared.
4. Choose New from the Script pop-up menu to open a new score script window.
   The lines `on exitFrame` and `end` automatically appear when the new script window opens. For this script, you need to replace the line `on exitFrame` with the phrase `on enterFrame`. (The phrase `on exitFrame` has the script run when the playback head leaves the frame. The phrase `on enterFrame` has the script run when the playback head enters the frame.)

5. Select the phrase `on exitFrame` and then type:

```
on enterFrame
  pause
```

When you are done, the script should look like this:

```
on enterFrame
  pause
end
```



BasicNav Score

Script: 29 on exitFrame

```
on exitFrame
 pause
```

Ink: Copy

Anti-Alias: Off

Display: Cast

Score Script 29

```
on exitFrame
  pause
end
```

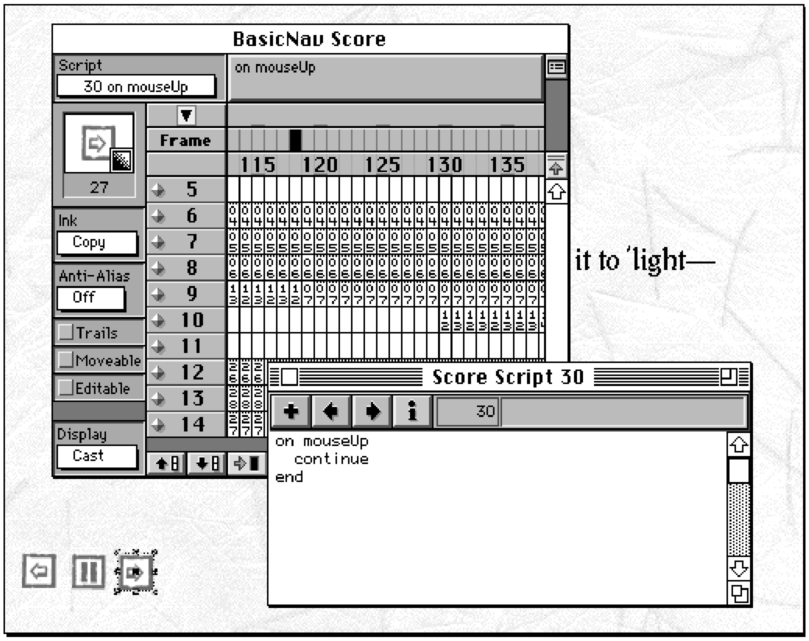
6. Press Enter or click the close box to enter the script.
The script is assigned to frame 118. Whenever the movie plays and enters frame 118, the script runs and the movie pauses.

To test the pause script you wrote, rewind and play the movie. Notice that the movie pauses at frame 118.

Typically, when you include a pause, you also include a `continue` script that tells the movie when to resume playing. For the “BasicNav” movie, you will assign a `continue` script to the forward arrow sprite in frame 118. The sprite script overrides the script of the cast member in any cells that the sprite script is assigned to.

To add the `continue` script:

1. Select the forward arrow sprite in channel 14 of frame 118.
2. Choose New from the Script pop-up menu at the top of the score to open a new script window.
   The lines `on mouseUp` and `end` automatically appear when the new score script window opens.
3. Type `continue`.



```
on mouseUp
  continue
end
```

4. Press Enter or click the close box to enter the script.
   The script is assigned to the forward arrow sprite in frame 118.

To test the scripts you created:

1. Rewind the movie and open the message window.
2. Play the movie.
   Notice that at frame 118, the movie pauses. The message window reports that the pause script ran.
3. Click the forward arrow.
   Notice that the movie resumes playing. The message window shows that the script assigned to the forward arrow in frame 118 ran when you clicked the sprite.

You just used the `pause` and `continue` commands to pause and continue the movie.

The `pause` command pauses the movie by stopping the playback head when the command is issued. Because the `pause` command is in the script channel, the `pause` command always runs whenever the movie enters the frame without relying on the user to do anything.

The `continue` command makes a paused movie resume playing. To let the user control when the movie resumes, you can assign the `continue` command to a sprite on the stage. That way, the `continue` command doesn't run until the user clicks the sprite.

A script assigned to a sprite overrides the script assigned to the sprite's cast member. The result in this case is that clicking the arrow in frame 118 has a different result than clicking the arrow in the rest of the movie. Assigning the continue script to the forward arrow's sprite rather than the cast member in this case is a good use of a sprite script, because the arrow uses the `continue` command in frame 118 only.

Returning to the same location

Sometimes, you want the movie to jump to a different frame or a separate movie and then return to the frame it left from. For example, in a scientific title, you might want to jump to a movie segment that explains a term and then return to your original location.

You could accomplish this by using `go to` statements. However, to return to the original frame, you would need to include Lingo that remembers the original frame number. Instead, you can accomplish the same thing with the `play` and `play done` commands.

Scene 11b of *Noh Tale to Tell*, in which Rokujo tells about a visit from Genji, is actually a separate movie that Director can branch to at the end of scene 10. The movie containing scene 11b is in the movie "NTBranch." As an exercise, re-create this Lingo in the "BasicNav" and "BNOM" movies in the Learning Lingo folder.

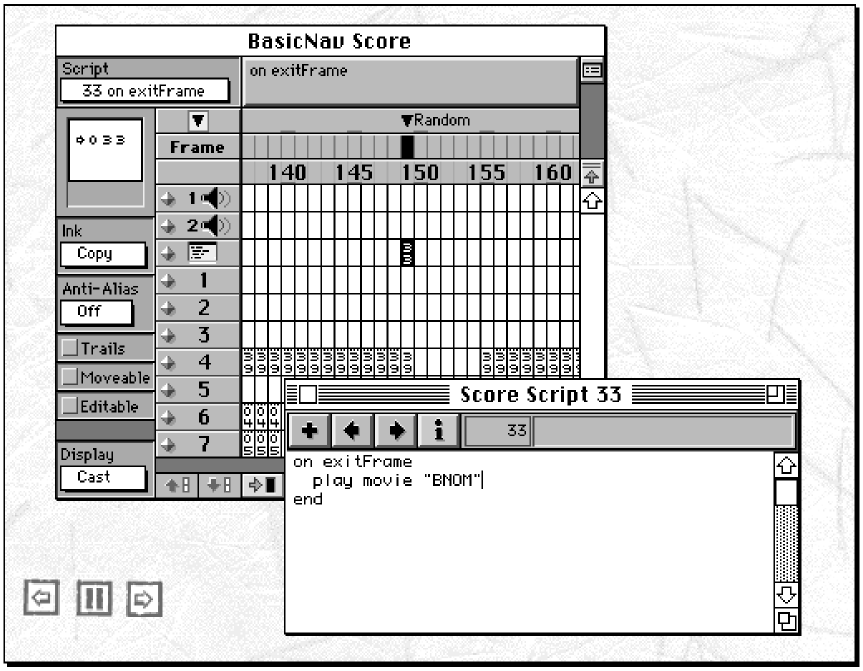
The finished version of *Noh Tale to Tell* contains Lingo that decides at random whether the movie branches or not. In this section, you write only the Lingo that branches. For an explanation of the Lingo that decides whether to branch, see the section, "Scripts that make choices," in Chapter 3.

First, play the unfinished version of the movie:

1. Open the movie "BasicNav" in the Tutorials: Learning Lingo: Storybook folder.
2. Open the score and play the movie.
3. Click the forward arrow at the end of scene 9.
   The movie plays to the end of scene 10, which is frame 148.
4. Stop the movie.

To write Lingo that has the movie branch to "BNOM":

1. Select the script channel in frame 149 of the score.
2. Choose New from the Script pop-up menu.
   A new script window appears.

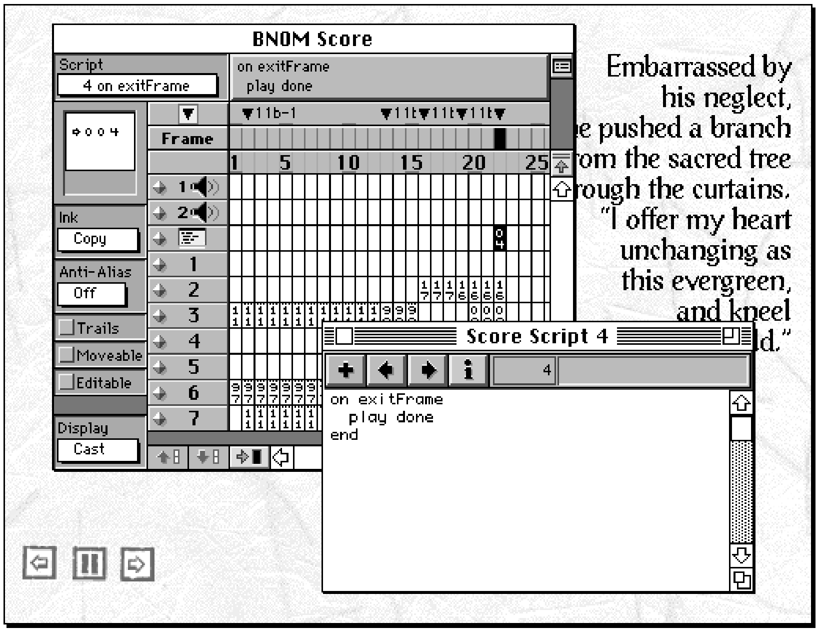


```
on exitFrame
  play movie "BNOM"
end
```

3. After the line on exitFrame, type:
   `play movie "BNOM"`
4. Press Enter or click the close box to enter the script.
   The new script replaces the frame script you assigned to this frame earlier in the section, "Creating loops."
5. Press Command-S to save the changes you made.

Next, add the `play done` command to the end of the movie "BNOM":

1. Open the movie "BNOM" in the Tutorials: Learning Lingo: Storybook folder.
2. Open the score.
3. Select the script channel in frame 30, the last frame of the movie.
   A new score script window appears.


BNOM Score
Script
4 on exitFrame
Frame
Ink
Copy
Anti-Alias
Off
Trails
Moveable
Editable
Display
Cast
Score Script 4
4
Embarrassed by his neglect, he pushed a branch from the sacred tree through the curtains. "I offer my heart unchanging as this evergreen, and kneel ...ld."

```
on exitFrame
  play done
end
```

4. Type `play done` after the line `on exitFrame`.
5. Press Enter or click the close box to enter the script.
6. Press Command-S to save the changes you made.

To see the effect of the `play` and `play done` commands you entered:

1. Open the modified movie “BasicNav.”
2. Open the message window and turn on the Trace checkbox.
3. Play the movie.
   The message window display and the playback head show how the movie plays to frame 149 and then branches to the movie “BNOM.”

   If you have modified the movie since you last saved it, the dialog box that asks whether you want to save changes appears when the movie branches to “BNOM.” When you branch between movies that you are still authoring, Director asks whether you want to save changes to the movie you are leaving.
4. Click Save.
   The movie “BNOM” plays until the `play done` command is encountered and returns to frame 149 of the movie “BasicNav.”

You just used the `play` and `play done` commands to have the movie branch to another movie and then return to the original frame. The `play` command doesn’t need you to specify where to return—it remembers the original frame so that the movie can return to it when the `play done` command is encountered or the sequence is finished.

The `play done` command is necessary when you want to return to the original frame before the sequence finishes. The `play done` command in the script you just wrote for “BNOM,” was optional, because it was at the end of the movie anyway.

The `play done` command used by itself sends the playback head to the beginning of the movie when the `play done` command is encountered. This sometimes happens when you are authoring and open a segment that contains the `play done` command without having played the segment that contains the `play` command. When this happens, open the movie you would otherwise return to by choosing Open from the File menu.

The `play` command is useful when:

- The movie you want to play does not have instructions about where to return.
- You want to play several movies sequentially from a single script. When one movie finishes, control returns to the part of the script that issued the `play` command.
- You want to put a sequence inside another and easily return to where you were in the outer sequence.
- You want to jump to one loop from several different locations.

Depending on the parameters you include after the `play` command, the playback head can jump to a different frame, another movie, or a specific frame in another movie.

You can specify these places in several ways:

To jump to	Use	Examples

A different frame	the frame number or marker label	`play "Help"`
		`play frame 60`

The beginning of a another movie	the word `movie`, followed by the movie name	`play movie¬`
		`"Noh_Tale"`

A specific frame in another movie	the word `frame` followed by a frame identifier, then followed by the word `movie`, and the movie name	`play frame 15¬`
		`of movie¬`
		`"Noh_Tale"`
		
		`play frame¬`
		`"Memory" of¬`
		`movie "Noh_Tale"`

You can also use expressions or variables to specify the movie's name, frame number, or marker label. Chapter 3, "Concepts," tells you more about what variables are and how to use them.

Note that all the ways that can specify a location also work with the `go to` command.

Chapter 3

Concepts

This chapter covers scripting language concepts used to apply some of Lingo's more powerful features.

It tells you about:

- The different types of scripts
- Events and messages, how handlers respond to them, and strategies for placing handlers
- If–then logic structures that make choices depending on conditions and events
- Variables and arguments and how they are used to keep track of and share values in a movie
- The order in which Lingo flows, that is, how it follows a certain order when executing a series of statements
- Syntax rules for writing scripts and expressing numbers and text in a way that Lingo can understand.

Types of scripts

A script's type is determined by where it is placed in the movie. As you will see in the sections "How Director responds to messages" and "Strategies for placing handlers" later in this chapter, the type of script in which you place Lingo can affect the script's behavior.

You can assign scripts to different places in the score and cast. The title bar at the top of the script window tells the type of script in the window. You can turn a movie script into a score script or turn a score script into a movie script by choosing from the Type pop-up menu in the Script Cast Member Info dialog box.

The following sections briefly describe the different types of scripts that you can write in Director.

Primary event handlers

A primary event handler is a script that is available at any time or place in the movie. You have to explicitly define a primary event handler whenever you want it to be available in the movie and explicitly turn it off when it is no longer appropriate.

Use the following elements to define a primary event handler:

- `keydownScript`
- `mousedownScript`
- `mouseupScript`
- `timeoutScript.`

Create a primary event handler by setting one of these elements equal to the script you want. For example, the statement:

```
set the mousedownScript to "go to frame 20"
```

creates a primary event handler that sends the playback head to frame 20 anytime the mouse button is pressed, regardless of where on the stage the click occurred.

When you want to turn off the primary event handler, set the primary event handler to `EMPTY`. For example, to turn off the `mouseDown` script `set the mouseDownScript to "go to frame 20"`, use the statement s`et the mousedownScript to EMPTY`.

Sprite scripts

Sprite scripts are attached to a sprite cell or cells. They require keyboard or mouse input to be activated. A sprite script takes precedence over a script assigned to the sprite's cast member, if it has one.

Scripts of cast members

A script of a cast member is assigned to a cast member and runs when the cast member is clicked. A cast member script is useful when you want the cast member to always have the same script attached, regardless of where the cast member appears in the score.

Unlike movie scripts and score scripts, scripts of cast members do not appear in the cast window. You open them by selecting a cast member in the cast window and then clicking the script button or by clicking Script in the cast member's Cast Info dialog box. Cast members that have scripts assigned to them display a black "L" in the lower left corner of their thumbnails in the cast window.

Frame scripts

Frame scripts are assigned to the script channel in the score. A frame script can be activated by the appropriate event whenever the playback head is the script's frame. Frame scripts are a good place for scripts that you always want to run for an individual frame, without requiring input from the user.

Movie scripts

Movie scripts are not explicitly assigned to a sprite or to the script channel but are available to the entire movie while the movie plays.

Movie scripts can control what happens when a movie starts, stops, or pauses. You can define more than one handler in a movie script. Handlers in a movie script can be called from other scripts in the movie as the movie plays.

Introducing events, messages, and handlers

Lingo uses events, messages, and handlers to determine what occurs in a movie and then carry out the proper instructions.

The following example demonstrates how Lingo sends a message that an event occurred and activates the appropriate handler that contains the instructions you want executed.

First, create a test movie by doing the following:

1. Create a new movie.
2. Select frame 1 in the script channel. Then choose New from the Script pop-up menu to open a new window.
   A new score script window appears.
3. Following the line `on exitFrame`, type `go to the frame` and then click the close box or press Enter.

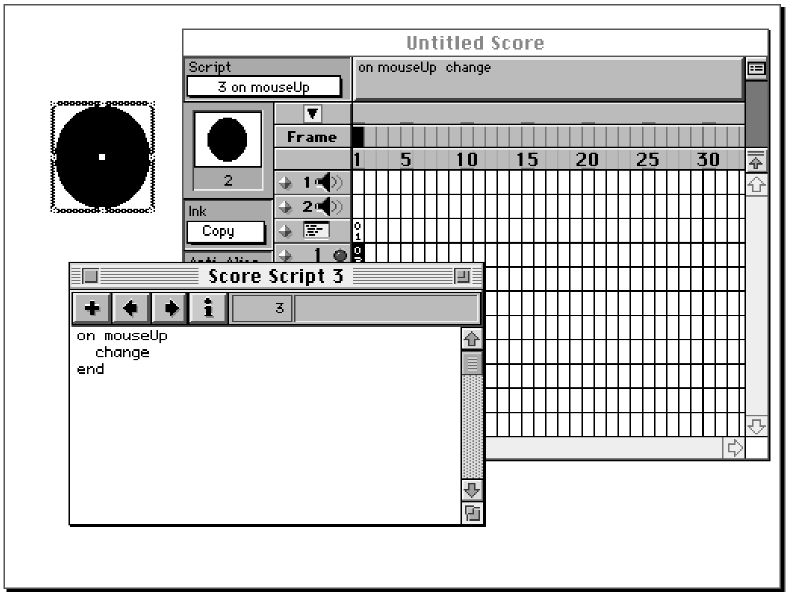
You just created a simple movie that loops in the first frame. Now add a sprite to which you will attach a handler:

1. Select the cell in frame 1 of channel 1.
2. Use the circle tool from the tools window to draw a circle.

Now write handlers for the circle sprite:

1. With frame 1 in channel 1 still selected, choose New from the Script pop-up menu.
   A new score script window appears.
2. Type the following handler in the script window:

```
on mouseUp
   change
end
```



```
on mouseUp
  change
end
```

3. Click the plus sign button in the script window to enter the current script and open a new movie script window.
4. Type the following handler in the new script window:

```
on change
   puppetSprite 1, TRUE
   set the foreColor of sprite 1 to ¬
   random (256) - 1
   updateStage
end
```

5. Press Enter or click the close box to enter the script.

To see what the handlers you just wrote do:

1. Open the message window and turn on the Trace option.
2. Rewind and play the movie.
3. Click the circle several times.
   The circle changes color when you click it. (It is possible that sometimes the circle turns the same color as the stage, in which case it seems to disappear until the color changes again.)
4. Press Command-S to save your movie. Name it whatever you like.

You just wrote a handler named `change`. The handler runs whenever you click the sprite in channel 1—the sprite that you attached the handler name to.

Some of the Lingo used in the handler script is probably new to you. You will learn more about this Lingo later, but briefly you wrote a script that:

◆ Puts the sprite in channel 1 under direct control of Lingo by using the `puppetSprite` command.

◆ Changes the sprite's color to a color chosen at random from the palette. The phrase `set the forecolor of sprite 1` followed by a number changes the sprite to the color that the number indicates on the current palette. The element `random` generates a random number. In this case, the quantity `(256)` specifies a number between 1 and 256. Therefore, the expression `random(256) - 1` gives a number between 0 and 255.

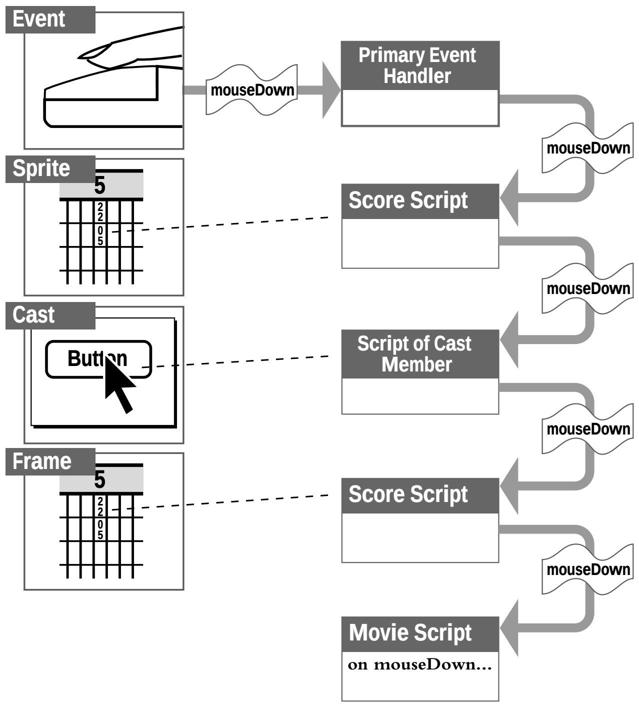
◆ Redraws the stage by using the `updateStage` command. Normally, Director redraws the stage only when the playback head enters a new frame. The `updateStage` command redraws the stage even though the playback head isn't moving.

Describing events

Events are occurrences in a movie. Some common types of events are clicking sprites, pressing keyboard keys, starting a movie, entering a frame, exiting a frame, or generating a certain result from a script.

Whenever an event occurs, Director generates a message describing the event. The message is sent to a series of objects. At the first object that has a handler with the same name as the message, Director executes the instructions in the handler. After the handler is executed, the message is no longer passed to other objects unless the handler includes an explicit instruction that the message be passed on.

When no such handler is found, the message travels to the next possible object until all possibilities are exhausted. If no script is found, the message is ignored. As an example, the following figure shows the series of objects that would receive the message that the mouse button was clicked.



Director has built-in message names for the common events that occur in a movie. These are the built-in messages available in Lingo:

Message	Event that occurred
`enterFrame`	Playback head entered the current frame
`exitFrame`	Playback head exited the current frame
`idle`	No event occurred while the movie is playing
`keyDown`	A key was pressed
`keyUp`	A key was released
`mouseDown`	A mouse button was clicked
`mouseUp`	A mouse button was released
`startMovie`	The movie started
`stopMovie`	The movie stopped
`timeOut`	A specified amount of time passed without a specified event occurring

You can have Lingo respond to these events by writing a handler that uses the event name as the handler name and placing the handler at a location that the message is sent to. For example, you can attach the following script to a sprite:

```
on mouseUp
  go to marker (1)
end
```

When you click and release the mouse button over the sprite, Director automatically generates the message `mouseUp` and starts searching for an `on mouseUp` handler. Because Director first looks for a sprite script assigned to the sprite you clicked, the handler in this example is the first one that Director finds and executes.

You can also define your own messages and corresponding handler names. In the test movie you just created, the term `change`—which you attached to the sprite script—is sent as a message whenever you click the sprite. After it is sent, the message looks for a handler that has the same name as the message. In this case, the handler was the handler you wrote as `on change`.

A statement that calls another script, another handler, or the statement's own handler is referred to as a calling statement. In the test movie's handler:

```
on mouseUp
  change
end
```

the statement `change` is a calling statement.

The `on mouseUp` handler is activated when it receives the `mouseUp` message.When Lingo encounters the term `change` in the script, it looks through a series of locations for a handler called `on change`. Provided that the handler is in the series of places Lingo looks, Lingo executes the first `on change` handler, if any, it finds.

How Director responds to messages

Lingo can send messages to cast members, sprites, cells, frames, windows, the computer, and the movie itself. An object intercepts a message when it has a handler script for the message unless the `pass` command explicitly sends the message on to the next object in the message hierarchy.

Messages follow a set order of objects when searching for a handler. In general, the sequence is primary event handler, sprite script, script of a cast member, frame script, and movie script.

Not all messages go to all these handlers. This is why understanding where specific types of messages are sent is important for deciding where to place scripts. The following sections, "Messages to primary event handlers" and "Messages to objects," describe where specific types of handlers go.

Messages to primary event handlers

When an event occurs, the message describing the event is first sent to a primary event handler. If a primary event handler exists for the message, the script is executed and the event is passed on to other objects, unless you explicitly stop the message by including the `dontPassEvent` command in the script. For example, when an `on mouseDown` primary event handler exists and the user presses the mouse button, the `on mouseDown` script is executed and then the `mouseDown` message passes to the next in the series of objects that can respond to a `mouseDown` message, unless the script includes the `dontPassEvent` command. (Think of the `dontPassEvent` command as equivalent to saying "don't pass the event on to the next location.")

This sample `on mouseDown` primary event handler has the movie display a notice whenever the mouse button is pressed any time the movie is playing. Because the handler doesn't include the `dontPassEvent` command, the `mouseDown` message then passes on to other scripts:

```
on mouseDown
  alert "Thanks for using the mouse"
end
```

Occasionally you want to prevent a primary event handler from executing (such as when the user holds down a modifier key during the event). In this case, use the `dontPassEvent` command to intercept the primary event handler.

For example, a regular `mouseDown` event causes Lingo to look for the sprite being clicked and then, if a sprite is clicked, to activate the sprite's script. Suppose you want to prevent the regular script from executing when the Option key is pressed when the sprite is clicked. You could use the following handler:

```
on mouseOption
  if the optionDown then
    dontPassEvent
  end if
end mouseOption
```

Activate the handler with the `mouseDown` primary event handler:

```
set the mouseDownScript to "mouseOption"
```

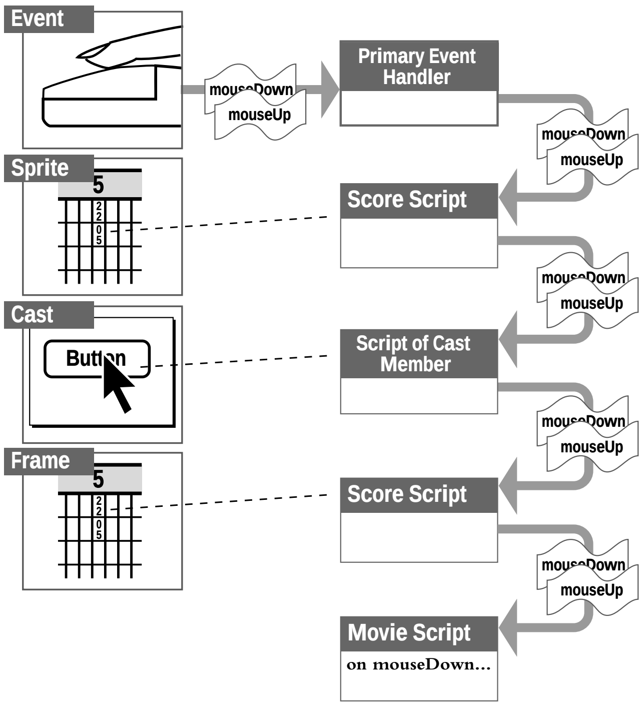
With the `mouseOption` handler, the mouse event also tests whether the Option key is being pressed at the same time as the mouse button. When it is not, the regular `mouseDown` event sequence is executed. When the Option key is pressed, the `dontPassEvent` command prevents the execution of the `mouseDown` primary event handler.

The `dontPassEvent` command applies only to the specific case where it is used. It does not create a new global primary event handler. Other event messages still pass to the proper handler. In other words, it doesn't have to be turned off.

Messages to objects

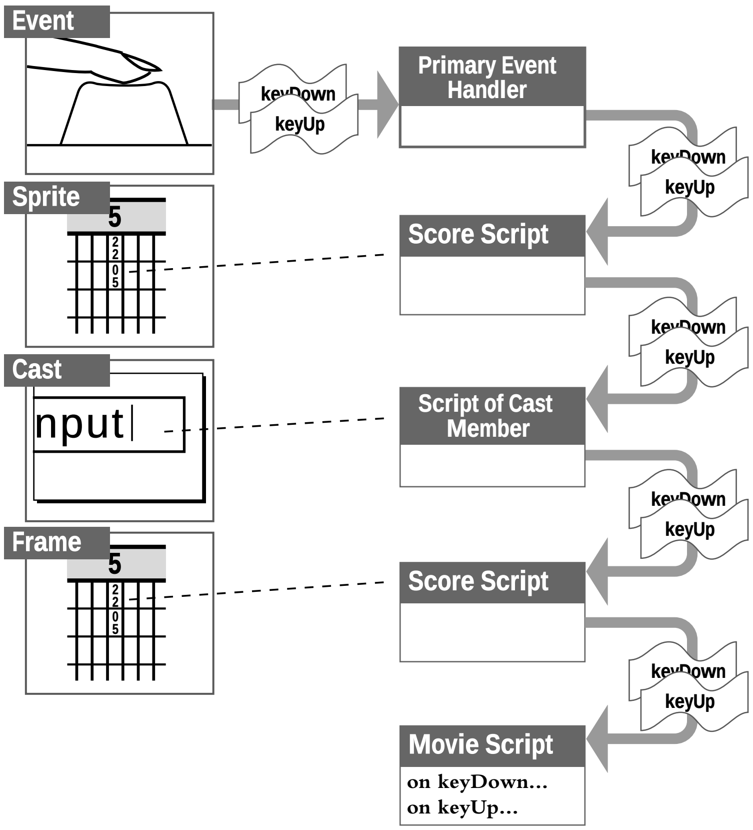
When an event has no primary event handler defined for it or the primary event handler doesn't include the `dontPassEvent` command, the message is sent to a series of objects. The order of objects depends on the individual message. The following illustrations show the order for built-in system messages. Lingo stops at the first one of these that has a handler defined for the message unless the script explicitly uses the `pass` command to tell Lingo to go on. When the handler contains the `pass` command, Lingo executes the statements in the handler and then passes the message on to any other objects that might also have a handler for the message.

The `mouseDown` and `mouseUp` messages are sent to a series of objects, as shown in the following figure:



- When the `mouseDown` or `mouseUp` occurs over a sprite, the message goes first to the sprite script, then to the script of the cast member, to the frame script, and finally to movie scripts. When more than one movie script contains a handler for the event, the handler in the script that has the lowest cast number is executed.
- When the `mouseDown` or `mouseUp` doesn't occur over a sprite, the message goes to the frame script and then to the movie script. When more than one movie script contains a handler for the event, the handler in the script that has the lowest cast number is executed.

The `keyDown` and `keyUp` messages are sent to the series of objects shown in the following figure:



- If the `keyDown` or `keyUp` occurs when the cursor is in an editable text sprite, the message goes first to the sprite script, then to the script of the cast member, to the frame script, and finally to movie scripts. When more than one movie script contains a handler for the event, the handler in the script that has the lowest cast number is executed.
- If the `keyDown` or `keyUp` doesn't occur when the cursor is in an editable text sprite, the message goes to the frame script and then to the movie script. When more than one movie script contains a handler for the event, the handler in the script that has the lowest cast number is executed.

The `enterFrame`, `exitFrame`, `idle`, and `timeOut` messages are sent to a frame script and then a movie script.

If the current frame has no frame script when the event occurs, the message goes to movie scripts.

The `startMovie` and `stopMovie` messages go directly to movie scripts only.

Messages that call handlers that you define yourself—for example, the `change` handler in the sample movie—can be sent from any script in Lingo. When this happens, Lingo searches for a corresponding handler in the script from which it was called. If and only if no handler is found, Lingo searches for a handler in one of the movie scripts.

If more than one movie script handles the same message, Lingo searches the movie scripts according to their order in the cast window, starting with the lowest numbered cast member. Lingo uses the first handler it finds; other handlers for the message are ignored.

The advantages of handlers

Handlers simplify script writing: you write a handler once and then call it from different places in the movie by entering the handler name. You can easily reuse the handler without copying or rewriting large amounts of text. In addition, when you revise the handler, you revise every instance where the handler is used in the movie. You don't need to repeatedly revise the handler in each place that can call it.

▶ **Tip** *When you develop handlers you want to reuse in other movies, you can create a library of handlers as scripts in a shared cast and reuse them in many movies.*

Where to place handlers

You can put handlers in score scripts or movie scripts, or assign them to a cast member:

- Handlers in score scripts or movie scripts can be called from any script in the movie.
- Handlers in scripts assigned to a cast member can be called only by an event that involves that cast member.

You can define as many handlers as you want within one script. It's a good idea to group related handlers in a single place, though, for easier maintenance.

To call a handler in a score script or movie script, place the handler name in the script that you want to have call the handler. This is what you did when you entered the name of the handler as a script assigned to sprite 1 in the section "Introducing events, messages, and handlers."

You can also place calling scripts inside other handlers. When the called handler stops executing, the handler that called it resumes.

Defining handler names

A handler name must meet these requirements:

- Starts with a letter
- Includes alphanumeric characters only (no special characters or punctuation)
- Is one word—no spaces are allowed
- Is not the same as a Lingo element.

▶ **Tip** *Using Lingo keywords for handler names can create confusion. Although it is possible to explicitly replace or extend the functionality of a Lingo element by using it as a handler name, this should only be done by advanced users.*

When you have more than one handler with similar functions, it is useful to give them names that have similar beginnings so that they appear together in alphabetical listing such as the listing given by the Find Handler command in the Text menu.

Strategies for placing handlers

Director follows a sequence of locations in which it searches for a handler that corresponds to a message. The locations are different for different messages, and the message typically doesn't pass on to other locations after the first script intercepts it.

The order of objects that messages travel through is important for deciding where to place handlers. You want to put handlers where Lingo can find them. Here are a few important points about placing handlers:

- Place `on startMovie` and `on stopMovie` handlers in movie scripts only. Lingo can't find these handlers if they are attached anywhere else.
- Place `on enterFrame`, `on exitFrame`, `on idle`, and `on timeOut` handlers in a frame script when you want the handler to run in that frame only. When you want the same handler to run in all frames, you can place the handler in a movie script.

  You can still assign an individual frame a different handler from the one in the movie script. Because these messages look for frame scripts before they look for a movie script, the handler in the frame script executes before the handler in the movie script and intercepts the message. Of course, you could still have the message go on to the movie script after the handler in the frame script is executed by using the `pass` command in the handler in the frame script.
- Place `on mouseDown`, `on mouseUp`, `on keyDown`, and `on keyUp` handlers in a sprite script when you want to assign the handler to that sprite only. When you want the same handler to be available for every instance of a cast member, you can place the handler in a script of the cast member.

When a cast member has a handler assigned, you can still assign a sprite a different handler. Because these messages look for sprite scripts before they look for a script of a cast member, the handler in the sprite script executes before the handler in the script of the cast member. Of course, you could still have the message go on to the handler in the script of the cast member after the handler in the sprite script is executed by using the `pass` command in the handler in the sprite script.

Putting one of these handlers in a frame script or movie script has the handler execute any time the corresponding event occurs in the frame or during the movie, unless there is a sprite or cast member script that intercepts the message first.

Describing conditions

Determining whether or not a condition exists is a common task for scripts. Fortunately, the choices are simple: a condition exists or it doesn't:

- To say that a condition exists, Lingo uses the term `TRUE` or the number `1`.
- To say that a condition doesn't exist, Lingo uses the term `FALSE` or the number `0`.

For example, the phrase `the moveableSprite of sprite 1 = TRUE` is one way that Lingo states that sprite 1 is draggable. An equivalent expression, using `1` instead of `TRUE`, is `the moveableSprite of sprite 1 = 1`. (Using Lingo to make sprites draggable is described in the section "Making sprites draggable," in Chapter 5.)

Lingo can also use the elements `TRUE` and `FALSE` to set a condition. For example, the statement:

```
set the moveableSprite of sprite 1 to TRUE
```

makes sprite 1 draggable. The statement `set the moveableSprite of sprite 1 to FALSE` makes sprite 1 not draggable. If you want to use numbers instead of the terms `TRUE` and `FALSE`, write the statements `set the moveableSprite of sprite 1 to 1` and `set the moveableSprite of sprite 1 to 0`.

Scripts that make choices

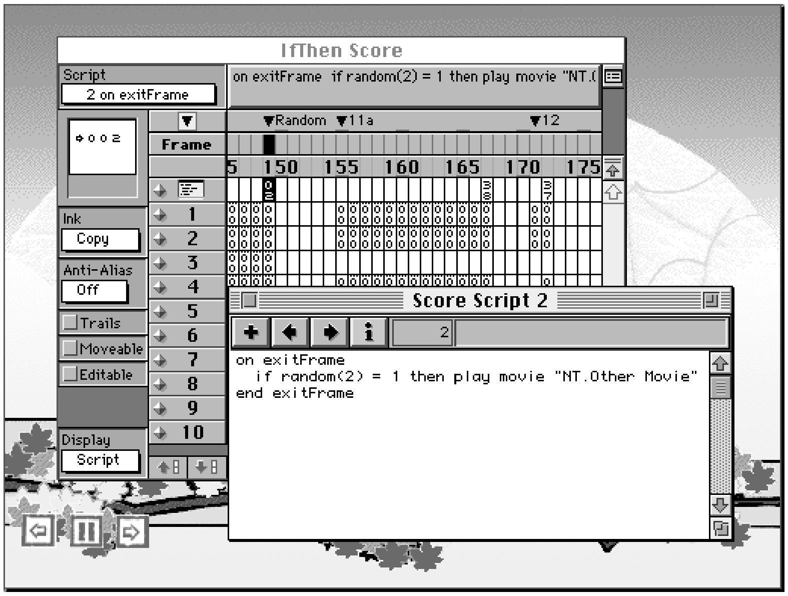
In everyday conversation, you often say that an action will occur if a certain condition exists. For example, the statement "if the light bulb is off, I will turn it on" bases the action "I will turn it on" on whether the condition "the light bulb is off" is true.

This section describes several ways that Lingo statements can be combined to make actions in the movie dependent on movie conditions.

An example of Lingo making a decision is at the end of scene 10 of the sample movie *Noh Tale to Tell*. At this point, Lingo decides whether to branch to the movie "NTBranch" based on a number generated at random. In the following exercise, you'll re-create the handler that performs this action.

To create a handler that makes an action dependent on a condition:

1. Open the movie "IfThen" in the Tutorials: Learning Lingo: Storybook folder.
   "IfThen" contains scenes 9 through 12 of the title *Noh Tale to Tell*.
2. Open the score and play the movie.
   The movie plays to scene 12 without branching.
3. Stop the movie.
4. Select the script channel in frame 149.
5. Choose New from the Script pop-up menu.
   A new score script window appears.



```
on exitFrame
  if random(2) = 1 then play movie "NT.Other Movie"
end exitFrame
```

6. Type the following handler in the score script window:

   ```
   on exitFrame
     if random(2) = 1 then play movie "ITOM"
   end
   ```

   The term `random` followed by a number is a Lingo element that generates a random number between 1 and the number you specify. The term `random(2)` generates either 1 or 2.

   The movie "ITOM," is the separate movie to branch to. It contains a modified version of "NTBranch" for this exercise. The movie "ITOM" already contains a `play done` command that returns the playback head to the original movie.

7. Rewind and play the movie several times.
   Sometimes the movie plays "ITOM" before playing scene 11a, but other times it skips "ITOM" and goes directly to scene 11a.

8. Close the movie.
   You do not need to save the changes you made.

Which scene plays depends on what `random(2)` generates:

- When `random(2)` generates 1, Lingo plays "ITOM."
- When `random(2)` doesn't generate 1, Lingo ignores the statement following `then` and the playback head goes to marker 11a.

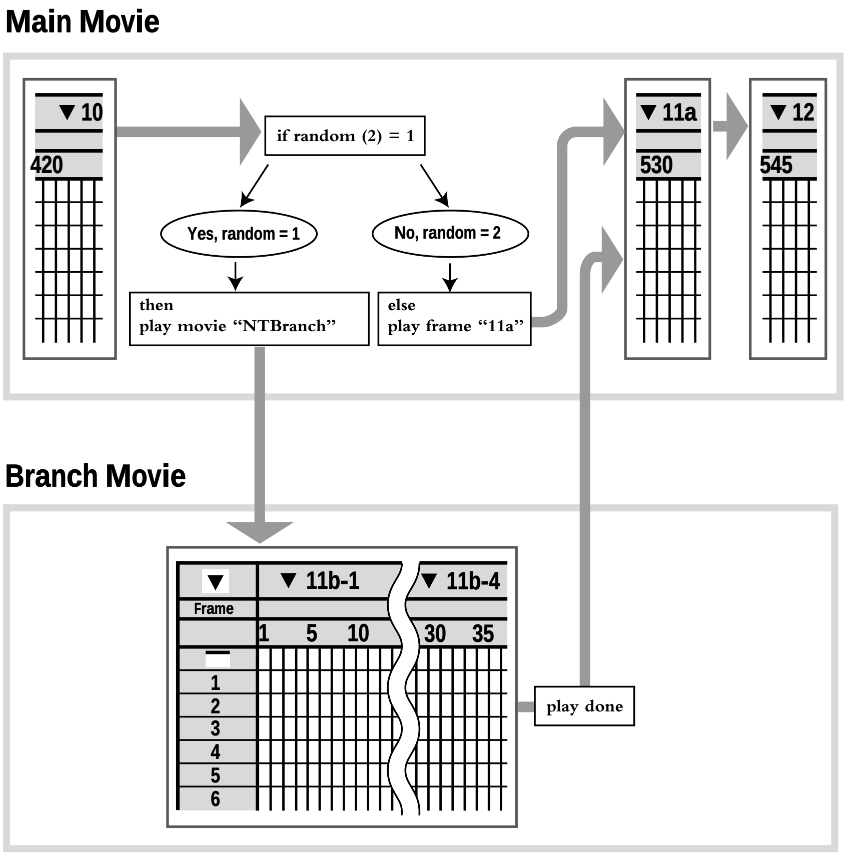
The `on exitFrame` handler includes an example of a Lingo statement that decides to perform an action depending on whether a condition exists. In this case, the action is playing the movie "ITOM." The required condition is that `random(2)` equals 1.

This handler is a simple example of a script structure that gives movies the flexibility to respond to changing conditions that occur when a movie plays. As the following sections explain, you can also create script structures that choose from more than one condition or keep repeating an action as long as a condition exists.

if…then…else structures

Using `if` with the elements `then` and `else`, you can have Lingo test whether a condition exists and respond accordingly. An everyday example of a condition you could test for is whether the light bulb is on or off. The statement `if sprite 1 intersects 2` is a similar test. Tests for a certain state usually begin with the Lingo element `if`.

When the condition exists, Lingo executes one set of statements; if the condition doesn't exist, Lingo executes a different set of statements. The following figure shows how a script branches one way or the other depending on whether the `if` statement is true or false.



This is a simple structure with one branch. Use this structure anytime you want to evaluate one condition and branch to one of two possible sets of statements.

When you want a script to branch to one of several possibilities, use a series of such tests.

For example, suppose you created an information kiosk about cities of the world. You might want to let the user go to more information about a specific city by clicking a familiar city landmark.

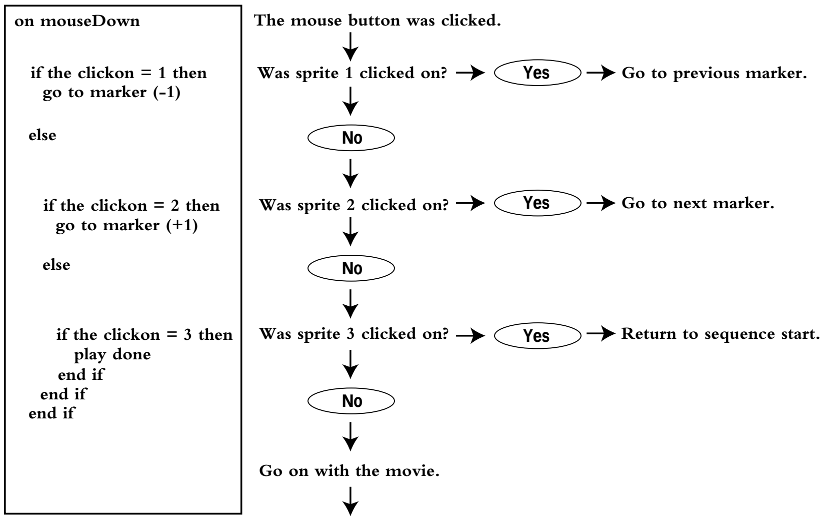
Every time the user clicks a landmark, the following handler would run through the series of tests (each starting with the Lingo element `if`) to check whether the click was on a sprite that was a landmark. (The phrase `the clickOn` is a Lingo element that indicates the number of the sprite that was clicked last. Only sprites that have a script assigned to the sprite or the sprite's cast member register a `clickOn`. Lingo perceives clicking sprites that have no script assigned as the same as clicking the stage.)

```
on mouseDown
  if the clickOn = 1 then
    go to "London"
  else if the clickOn = 2 then
    go to "Paris"
  else if the clickOn = 3 then
    go to "New York"
  else if the clickOn = 4 then
    go to "Tokyo"
  else
    nothing
  end if
end
```

As soon as one of the tests is true (the click occurred on the sprite whose number follows `the clickOn =`), the phrase `then go to` followed by a marker label sends the playback head to the specified frame of the movie.

Note *The previous situation is an example of a script that would probably best be used as a movie script. This is because the* `mouseDown` *event used to click the sprite has broad use and needs to be available throughout the movie.*

This is how scripts that test for several possibilities can branch:



Lingo evaluates a series of tests like this in order. The first time a condition is found to exist, Lingo performs the action that follows `then`. Otherwise Lingo goes on to the next test. Finally, if the click wasn't on one of these sprites, Lingo reaches the end of the set of statements and nothing happens.

▶ ***Tip*** *To optimize your script's performance, test for the most likely conditions first.*

When writing if–then scripts, place the statement following `then` in the same line as `then` or place it on its own line by inserting a Return after `then`. For example, the statement:

```
if the clickOn = 1 then go to "Paris"
```

is equivalent to:

```
if the clickOn = 1 then
  go to "Paris"
end if
```

However, you must include an `end if` statement at the end of the if–then structure if its lines include a carriage return.

The script window automatically indents the statement that follows `then` when you enter a carriage return. This indentation helps organize the scripts visually and makes it easier to see the blocks of Lingo involved in the if–then structure.

Repeating an action

Using a `repeat while` statement, you can repeat a set of instructions as long as a specific condition exists. For instance, if you want a movie to beep continually whenever the mouse button is held down, use the following statements:

```
repeat while the mouseDown
  beep
end repeat
```

Lingo continues to loop through the statements inside the repeat loop until the condition is no longer true or one of the instructions sends Lingo outside the loop. In the example, Lingo exits the repeat loop when you release the mouse button; the `mouseDown` condition is no longer true when this occurs.

Note *When executing a repeat loop, Lingo ignores any events that occur in the movie. As a result, statements in the loop can be processed faster than they would be otherwise. However, no other tasks can be performed during this time.*

If the condition is always true or you want the repeat to stop when some other event occurs, use `exit repeat` to send Lingo out of the repeat loop.

The following script uses `exit repeat` to make the movie beep while the mouse button is pressed unless the mouse pointer is over a specified sprite. When the pointer is over sprite 1, Lingo exits the repeat and stops beeping. (The Lingo element `the rollover` followed by a sprite number indicates whether the cursor is over the specified sprite.)

```
repeat while the mouseDown
  beep
  if rollover (1) then exit repeat
end repeat
```

If the condition you are testing is always true and there is no exit, usually the only way you can exit the loop is to press Command–period. Pressing Command-period stops the entire movie.

Note *Because Director ignores all other events when it is in a repeat loop, it isn't a good idea to use repeat while statements for delays. It is best to create delays by using the timer, putting a delay in the tempo channel of the score, or by adding extra frames to the score. See the entry for the timer property in the Lingo Dictionary for an explanation.*

Repeating a specific number of times

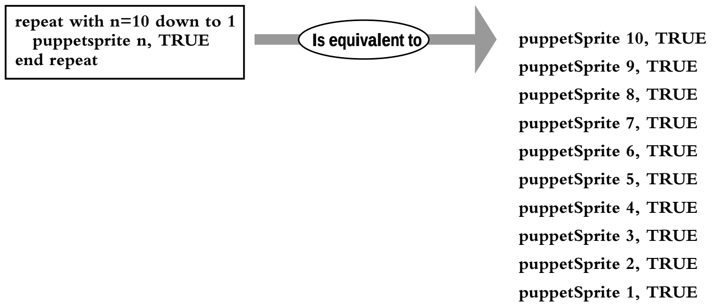
A `repeat with` statement repeats an operation for a specified number of times. The number of times to repeat is defined as a range following the `repeat with` element.

Suppose you want to change the foreground color of sprites 1 through 10. You can change the foreground color of a sprite using the Lingo statement `set the forecolor of sprite n to` followed by the new color's number on the palette. When you use this statement, replace `n` with the number of the sprite whose color you want to change. Because you must specify the sprite number in the statement, you could write a separate statement for each sprite.

Alternatively, you can use the following repeat loop to change the foreground color of sprites 1 through 10:

```
repeat with n = 10 down to 1
  set the forecolor of sprite n to 90
end repeat
```

The following figure shows how this repeat loop is equivalent to using ten separate statements to change the foreground color of ten sprites:



```
repeat with n=10 down to 1
   puppetsprite n, TRUE
end repeat
```

```
puppetSprite 10, TRUE
puppetSprite 9, TRUE
puppetSprite 8, TRUE
puppetSprite 7, TRUE
puppetSprite 6, TRUE
puppetSprite 5, TRUE
puppetSprite 4, TRUE
puppetSprite 3, TRUE
puppetSprite 2, TRUE
puppetSprite 1, TRUE
```

The loop repeats the statement `set the forecolor of sprite n to 90` ten times. The first time, `n` equals 10; the second time n equals 9; and so on until `n` equals 1. After the tenth time, the loop is complete and Lingo goes on to the next event.

Note *In actual practice, you should make the sprite a puppet before changing its foreground color. Puppets are explained in Chapter 4, "Working with Puppets."*

Updating values

Director remembers and updates values by using variables. As the name implies, a variable is an object in memory that contains a value that can be changed or updated as the movie plays. By changing the value of a variable as the movie plays, you can do things such as store information the user enters, track the points in a game, or record whether a specific event has happened.

The value assigned to the variable can be a whole number, a decimal number (such as 1.56), a character string (such as “xyz” or a person’s name), or the result of a calculation.

To experiment with variables, do the following:

1. Create a new movie.
2. Open the message window and type:

   ```
   put 5 + 5 into mySum
   ```
3. Press Return.
4. Type `put mySum` and then press Return.
   The message window displays the number 10, which is the sum of 5 + 5.
5. Now type `put mySum + 1 into mySum` and press Return.
6. Type `put mySum` and press Return.

   The message window displays the number 11, which is the sum of 10 + 1.
7. Press Return.

Notice that each time, the value that appears in the message window is different. In the first case, it was 10: the value of 5 + 5. In the second case, it was 11, the value assigned to `mySum` in the previous statements plus 1. However, you used the statement `put mySum` each time. After a variable has been assigned a value, the variable name, used alone, provides the most recent value given the variable.

You just wrote scripts that created a variable and assigned values to it. In this case, you made the `mySum` variable contain different values at different times.

▶ **Tip** *As in the Lingo Expo movies, it is a good habit (and common usage among Lingo users) to use variable names that indicate what the variable is used for. For example, the variable mySum indicates that the variable contains the sum of numbers.*

Assigning values to variables

You assign a value—such as a number or a string of text—to a variable with the `put ... into` command. For example, the statement `put "Mary" into theName` assigns the text string "Mary" to the variable `vName`.

Values assigned to a variable (in this case `theName`) can be generated in many ways. Some possible sources for variable values are text that the user types, the result of an arithmetic operation that Lingo performs, and the result of clicking a particular sprite.

For example, the following set of statements assigns one of three values to `theName`, including the value `EMPTY` if the user doesn't click either of the two specified sprites:

```
if the clickOn = 3 then
  put "Mary" into theName
if the clickOn = 4 then
  put "John" into theName
  else
    put EMPTY into theName
  end if
end if
```

In this example, the Lingo term `EMPTY` specifies an empty string that has no characters.

You can also use the `set` command to assign a value to a variable. For example, these statements are equivalent ways to assign "Mary" to the variable `theName`:

```
put "Mary" into theName
set theName = "Mary"
set theName to "Mary"
```

However, it is standard practice among Lingo programmers to use `put ... into` when assigning values to variables. The `set` command is typically used to specify properties, as in the statement `set the forecolor of sprite 2 to 90`.

Creating variables

A variable is created the first time you assign a value to it, which is also called initializing a variable. You can then use the variable in other expressions or change its value based on whatever criteria you want. How long a variable exists depends on whether it's a local or global variable.

▶ **Tip** *It's a good idea to always assign a variable a known value the first time you define it. This makes it easier to track and compare the variable's value as the movie plays.*

Local variables

A local variable exists only as long as the script in which it is defined is running. You can use a local variable in any script or handler. It is subsequently available only while that script or handler is being executed.

Any variable created in a handler or script that is defined without using the term `global` is automatically a local variable.

You can display all current local variables by using the `showLocals` command in the handler. This command can be used in the message window or in handlers to help with debugging. The result appears in the message window.

Global variables

A global variable exists and retains its value for as long as Director is running or until you issue the `clearGlobals` command. This gives you the opportunity to use the value assigned to a variable throughout the movie or among movies after the handler in which they were defined is finished executing.

For example, to use someone's name several times in your movie, you could establish a global variable that is assigned a name typed by the user at the beginning of the movie.

Global variables can be defined or assigned new values within a movie script, score script, or in the message window.

You make a variable in a movie script or score script into a global variable by using the term `global` before the variable name the first time you define it and in every handler that uses the global variable; variables that you declare in the message window are automatically global.

For example, the following statements makes `theName` a global variable and give it the value `Mary`:

```
global gName
put "Mary" into gName
```

This handler could use the same global variable and change its value to "John:"

```
on nameChange
  global gName
  put "John" into gName
end
```

▶ **Tip** *It is a good habit to start the names of all global variables with a small letter "g." This helps identify which variables are global when you examine Lingo code.*

A global variable that you've defined in a movie script or in the message window can be referred to, and the value in it changed or used, by any other script or handler.

Placing the `global` command in front of several variables establishes them all as global at the same time. The following statement defines three global variables:

```
global gName, gCapture, gHelp
```

When you define global variables this way, they are automatically initialized as empty or 0. To assign other values to them, use `set` or `put`.

You can display all current global variables and their current values with the `showGlobals` command in the message window.

▶ ***Tip*** *Because you usually want global variables to be available throughout the movie, it is usually good practice to declare global variables in the on startMovie handler. This ensures that the global variables are available from the very start of the movie.*

Handlers that return results

Handlers that return a result are sometimes called functions.

You can define handlers that do not require arguments yet still return some value. (Arguments, which let you pass values between objects in the movie, are discussed in the next section, "Using arguments to pass values.") For example, the following handler returns the current color of sprite 1.

```
on findColor
  return the forecolor of sprite 1
end
```

Even when you define a handler that returns a result but doesn't require arguments, you must still use the parentheses when you call the function from another script. For example, this statement in the message window would call the findColor handler and then display the result in the message window: `put findColor()`.

Using arguments to pass values

Arguments are placeholders that let you pass values to types of structures such as handlers and methods. You are already familiar with handlers; methods, which are similar to handlers, are used in XObjects and factories. For information about using methods, see Appendix B, "XObjects," and Appendix C, "Factories."

By using arguments for values, you can give the handler exactly the values that it needs to use at a specific time, regardless of where or when you call the script in the movie. Arguments can be optional or required depending on the situation.

You create arguments for a handler by putting them after the handler name. Multiple arguments are separated by commas. For example, the following handler, called `addThem`, adds two values it receives in the arguments `a` and `b`, stores the result in the local variable `c`, and uses the Lingo term `return` to send the result back to the original handler.

```
on addThem a, b
  -- a and b are argument placeholders
  set c = a + b
  return c
end addThem
```

You provide specific values (called parameters) for the arguments that the handler uses when you call the handler. Parameters can be any type of value, such as a number, a variable that has a value assigned, or a string of text. Parameters in the calling statement must be in the same order that they have in the first line of the handler and be surrounded by parentheses.

The following statement is a calling statement for the `on addThem` handler described above`: set mySum = addThem (4, 8`).

In this example, `4` corresponds to the argument `a` and `8` corresponds to the argument `b`. You can also use variables as parameters.

After the calling statement sends these parameters to the handler, the handler returns the value `12`, which corresponds to the variable `c` inside the `addThem` handler. The variable `mySum` is then set to `12`.

How Lingo flows

Lingo always executes statements in a script starting with the first statement and continuing in order until a statement tells Lingo to go somewhere other than the next statement or until the final statement is reached. Some statements that can send Lingo to somewhere other than the next statement are repeat loops, if–then–else structures, and handler names placed within scripts.

The order in which statements are executed affects the order in which you should place statements. For example, if you write a statement that requires some calculated value, you need to put the statement that calculates the value first. Or, as you did in the sample movie at the beginning of this chapter, if you write a statement that changes the display on the stage, the `updateStage` command needs to come after the statement.

You can take advantage of the order in which Lingo executes statements when you debug a script by tracing a script's progress in the message window as the movie runs. The most common use of this is to see at which line of a problem script an error message is generated. This is usually a good pointer to the problem.

Working with values

A value is any quantity assigned to a variable, parameter, or symbol. Some examples of values you encounter using Director are the name or number assigned to a variable, the number generated by a function such as `random()`, and whether some property such as a sprite being moveable is `TRUE` or `FALSE`. Use the `set` and `put` commands to test and set conditions. Use operators to evaluate and manipulate numbers and strings. The following sections describe both situations.

Testing and setting conditions

A property is any attribute of an object (such as a menu, cast member, or field) or of the computer that can have one of several settings. Some examples of properties used in Lingo are the cast member assigned to a sprite, the monitor's color depth, a sprite's location on the stage, and the time in Director's timer.

Lingo can test and set properties for the system, for sprites, and for text objects. Use the `set` and `put` commands to change and return the values of properties.

For example, the statement:

```
set the castNum of sprite 2 to 113
```

changes the cast member assigned to sprite 2 by setting the sprite's `castNum` property to a different cast number.

You can't set all properties. Some property values can only be read. Often these are properties that describe some condition that has already occurred. For example, the property `timeoutLapsed`, which indicates the length of time since the last `timeOut`, can only be read.

Default values and options for each predefined property are provided in the *Lingo Dictionary*.

Operators

Operators are elements that tell Lingo how to combine, compare, or modify the values of an expression:

- Arithmetic operators (such as `+`, `-`, and `*`)
- Comparison operators (for example, `>`, `>=`, and `=`), which compare two arguments
- Logical operators (`not`, `and`, `or`), which combine simple conditions into compound ones
- String operators (`&` and `&&`), which join strings of characters.

When two or more operators are used in the same statement, some operators take precedence over others in a precise hierarchy that Lingo follows to determine which operators to execute first. This is called the operator's precedence order. For example, multiplication is always performed before addition. However, items in parentheses take precedence over multiplication. For example, without parentheses, Lingo performs the multiplication in this statement first:

```
set total = 2 + 4 * 3
```

The result is 14.

When parentheses surround the addition operation, Lingo performs the addition first:

```
set total = (2 + 4) * 3
```

The result is 18.

The operators and their precedence orders are described in the following sections. Operators with higher precedence are performed first. For example, an operator whose precedence is 5 is performed before an operator whose precedence is 4. Operations that have the same precedence are performed left to right.

Arithmetic operators

Arithmetic operators add, subtract, multiply, divide, and perform other arithmetic operations. Parentheses and the minus sign are arithmetic operators.

Arithmetic operators

Operator	Effect	Precedence
`( )`	Groups operations to control precedence order	5
`-`	When placed before a number, reverses the sign of a number	5
`*`	Multiplication	4
`mod`	Modulo	4
`/`	Division	4
`+`	Addition	3
`-`	When placed between two numbers, performs subtraction	3

Comparison operators

Comparison operators compare two values and determine if the comparison is TRUE or FALSE.

You can compare strings or numbers with all the other comparison operators. In the case of strings, "greater than" (>) means "later in alphabetical order." String comparisons ignore upper- and lowercase.

Comparison operators

Operator	Effect	Precedence
`sprite... within`	TRUE if sprite 1 is entirely within sprite 2	5
`sprite... intersects`	TRUE if sprite 1 touches sprite 2	5
<	Less than	1
>	Greater than	1
<=	Less than or equal to	1
>=	Greater than or equal to	1
=	Equal to	1
<>	Not equal to	1
`contains`	True if string 1 contains string 2	1
`starts`	True if string 1 starts with string 2	1

Logical operators

Logical operators compare conditions and determine whether one condition exists, all conditions exist, or the conditions are different.

For example, the `or` operator allows you to create a condition which is `TRUE` if either of two or more conditions is `TRUE`:

```
if (number <= 100) or (time > 180) then
  set the text of cast 14 to "Do you want some help?"
```

Logical operators

Operator	Effect	Precedence
`not`	Inverts the value of a condition	5
`and`	Determines if all conditions exist	4
`or`	Determines if one or more conditions exist	4

▶ ***Tip*** *Surrounding elements of a logical comparison with parentheses makes the parts of the comparison easier to see, even though the parentheses aren't required.*

String operators

String operators join or concatenate strings together.

The & operator joins the text strings that appear before and after the & operator.

The && operator joins text strings but inserts a space between them.

For example, the statement:

```
set the text of cast 14 to "Hello," && theName
```

combines "Hello" and whatever name is assigned to the variable `theName` to create one string with a space between the two. If the value of `theName` was "John," the result would be "Hello, John"

String operators

Operator	Effect	Precedence
&	Concatenates strings	2
&&	Concatenates strings and inserts a space	2

Expressing literal values

A literal value is any part of a statement or expression that is to be taken exactly as it is rather than as a symbolic value. Literal values that you encounter in Lingo are text strings, integers, decimal numbers, cast member names, cast member numbers, symbols, and constants.

Each type of literal has its own rules for how it can be expressed. The following sections present the rules for each type of literal.

Text strings

Literal text strings are text that you want to treat as such instead of as a variable. Literal text strings must be enclosed within quotation marks. For example, in the statement:

```
put "Hello" into field "greeting"
```

“Hello” and “greeting” are both literal text strings. “Hello” is the actual text being put into a field; “greeting” is the actual name of the text cast member.

Similarly, if you test a text string, double quotation marks must surround each string, as in the following example:

```
if "Hello Mr. Jones" contains "Hello" ¬
then soundHandler
```

Lingo treats spaces at the beginning or end of a string as a literal part of the text. The following expression includes a space after the word to:

```
put "My thoughts amount to "
```

Integers

An integer is always a whole number, without any fractional decimal places.

Director works with integers between –2,147,483,648 and +2,147,483,647. Enter integers without using commas. Use a minus (`-`) sign for negative numbers.

Some Lingo commands and functions require a whole number argument that's within a given range. The requirements for specific Lingo elements can be found in the *Lingo Dictionary*.

Decimal numbers

A decimal number—sometimes called a floating-point number—is any number that includes a decimal point. (The `floatPrecision` property controls the number of decimal places used in numbers. See the `floatPrecision` entry in the *Lingo Dictionary* for information about setting the number of decimal places used for decimal numbers.)

You can use decimal numbers in Lingo scripts. You can also use exponential notation (for example: -1.1234e-100 or 123.4e+9).

You can convert an integer to a decimal number using the `float()` function. For example, the statement
`set theNumber = float(3)` changes the integer "3" to a floating point number. If Director is set to display numbers to four decimal places, the result would be 3.0000. (Use the `floatPrecision` function to specify the number of decimal places used to display integers.)

Cast numbers and names

Lingo can refer to bitmapped images, buttons, text, palettes, and sounds in the cast in one of three ways: by literal cast number (`cast 123` and so on), by cast number relative to another cast position (`cast 1 + 15`), by cast name (`"newFigure"`), or as a number of cast window slots relative to a named cast member (`cast "newFigure" + 10`).

You can add and subtract cast numbers in an expression to refer to different cast members. For example, `cast (1 + 1)` is equivalent to cast 2. The phrase `cast (211 +1)` is equivalent to cast 212.

If you name more than one cast member with the same name and then use the name in a script, Lingo uses the first (lowest numbered) cast member that has the specified name.

Symbols

A symbol is a type of data (like a string or other value) that begins with the pound sign (#). Symbols are useful because they can be obtained from memory more quickly than strings.

For example, the symbol `#Steve` in the statement:

```
put #Steve into userName
```

returns from memory more quickly than the string `"Steve"` in the statement:

```
put "Steve" into userName
```

Variables assigned symbols behave differently than other variables.

For example, the statement `put #Steve into userName` followed by `put userName` gives the result `#Steve`, because `#Steve` is the actual value of the variable.

The statement `put "My name is" && userName` in the message window displays the result `My name is Steve` because in this case Lingo looked for the text string assigned to the variable `userName`.

However, the statement `put userName + 1` gives a number, because in this case Lingo is performing an arithmetic operation and treats `userName` asa a numerical value.

Constants

A constant is a named value whose content never changes. For example, `TRUE`, `FALSE`, and `EMPTY` are constants because their value is always the same.

Refer to constants by their names. The constants `BACKSPACE`, `ENTER`, `QUOTE`, `RETURN`, and `TAB` let you refer to these characters in your scripts where needed. For example, to test whether the Return key is pressed, use the following expression:

```
if the key = RETURN
```

The elements of Lingo

Lingo communicates information through combinations of elements, which are equivalent to words. Like words in English, Lingo elements can be categorized by what these elements do and how they can be combined. All Lingo elements are included in the *Lingo Dictionary*, the Lingo menu, and in the online help.

Categories of Lingo elements include:

- Commands: instructions that cause something to happen while a movie is playing. For example, the command `go to` instructs the playback head to jump to a specified frame.
- Functions: elements that return some value. For example, the function `date` tells the current date set in the computer. The function `key` tells which key was pressed last. Other examples of functions are `abs()`, `random()`, and `sqrt()`.
- Keywords: reserved words that have a special meaning. For example, `the` is used to indicate that the next word is a property.
- Operators: symbols or words that change the value of one or more values. For example, the plus sign operator (+) adds two or more values together and produces a new value.
- Constants: elements that don't change. For example, the constants `FALSE` and `EMPTY` always have the same meaning.
- Properties: attributes of an object. For example, color depth is a property of a bitmap.

Elements can be combined into statements, which are equivalent to sentences in English. A statement is any valid instruction that Lingo can execute. When a statement is executed, its instructions cause Director to perform some action.

A script is any statement or group of statements that make up the contents of a movie script or score script in the cast window or a script of a cast member window. Handlers are the typical units, often called routines by programmers, found within scripts.

An expression is any part of a statement, meant to be taken as a whole. For example, `2 + 2` is an expression but is not a valid statement all by itself. The line `go to frame 23` is a statement—`go to` is the command, and `frame 23` is the value that the command requires in order to carry out the instruction.

Using Lingo's syntax

Like any language, Lingo has rules of grammar and punctuation that you must follow. The following are general rules that apply to all Lingo. Most Lingo elements also have their own individual requirements about elements that they must be combined with. If you want to know the rules for a specific Lingo element, refer to the description of the element in the *Lingo Dictionary*.

Parentheses

Some Lingo functions require parentheses. The ones that do are clearly marked as such in the *Lingo Dictionary*. When you define functions in your own handlers, you need to include parentheses in the calling statement.

You can use parentheses to override Lingo's precedence or to make your Lingo statements easier to read.

Character spaces

Words within statements are separated by spaces. Lingo ignores extra spaces, so you can put them in if you want to use them for better formatting and readability.

In strings of characters surrounded by quotation marks, spaces are treated as characters. If you want spaces in a string, you must put them in explicitly.

You can see Lingo that uses strings in Chapter 6, "Using the Keyboard & Mouse."

Upper- and lowercase letters

Lingo is not case-sensitive—you can use upper- and lowercase letters however you want. For example, the following statements are equivalent:

```
Set the hilite of cast "Cat" to TRUE
Set the hiLite of cast "cat" to True
set the hilite of cast "Cat" to true
SET THE HILITE OF CAST "CAT" TO TRUE
Set The Hilite Of Cast "Cat" To True
```

However, even though Lingo is not case-sensitive, it is a good habit to follow scriptwriting conventions, such as the ones that are used in this manual. That way, it is easier to quickly identify names of handlers, variables, and cast members when reading Lingo code.

Comments

Comments in scripts are preceded by double hyphens (`--`). You can place a comment on its own line, or after any statement. Lingo ignores any text following the double hyphen on the same line.

Comments can be anything you want: notes about a particular script or handler, or about a statement whose purpose might not be obvious. Comments make it easier for you or someone else to understand a procedure after you've been away from it for a while.

Use the Comment and Uncomment commands in the Text menu to speed your entering and removing comments.

▶ ***Tip*** *When debugging, it is sometimes useful to isolate one section of Lingo that you want to focus on by temporarily commenting out other statements or handlers.*

Optional keywords and abbreviated commands

You can abbreviate some Lingo statements. Abbreviated versions of a command are easier to enter but may be less readable than the longer versions. The `go` command is a good example. All the following statements are equivalent. The last one takes the fewest number of keystrokes.

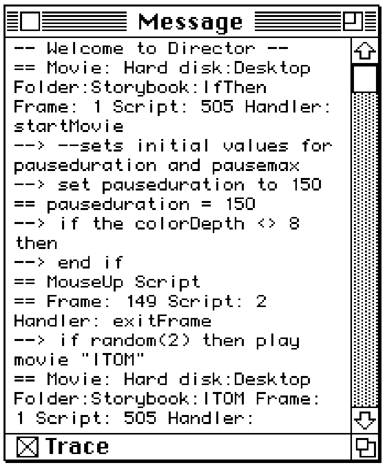
```
go to frame "This Marker"
go to "This Marker"
go "This Marker"
```

If a command can be abbreviated, the acceptable abbreviations are shown in the *Lingo Dictionary*.

▶ ***Tip*** *It is good practice to use the same abbreviations throughout the movie.*

The tracing symbols in the message window

This is a sample of the message window display that was made by turning on the Trace option and then running a script for a few moments.



```
-- Welcome to Director --
== Movie: Hard disk:Desktop
Folder:Storybook:IfThen
Frame: 1 Script: 505 Handler:
startMovie
--> --sets initial values for
pauseduration and pausemax
--> set pauseduration to 150
== pauseduration = 150
--> if the colorDepth <> 8
then
--> end if
== MouseUp Script
== Frame: 149 Script: 2
Handler: exitFrame
--> if random(2) then play
movie "ITOM"
== Movie: Hard disk:Desktop
Folder:Storybook:ITOM Frame:
1 Script: 505 Handler:
```

Notice that Lingo has placed different symbols at the beginning of each line.

- An arrow (-->) appears at the beginning of the line of the script being executed.
- A double equal sign (==) appears before the number of the script and the name of the handler when a handler starts being executed, any values displayed in the message window by the `put` command, and the frame number when the playback head enters a frame under direction from Lingo.
- The greater than (>) pointer shows the nesting level of the script. > for the first level, >> for a script called by a script, >>> for a script called by a script called by a script, and so on.

Chapter 4

Working with Puppets

This chapter introduces puppets—sprites, sounds, tempos, palettes, and transitions that you control from Lingo instead of from the score.

Puppets are dynamic. By using them, you can change a sprite's characteristics based on what the user chooses or the movie does. Because you can make these changes without leaving the current frame, puppets often let you use far fewer frames to achieve the features and behavior that you want.

The material in this chapter tells you:

- What puppets can achieve
- How to create a sample puppet
- Specific commands that make sprites, sounds, tempos, palettes, and transitions into puppets and the ways you can control them.

What puppets offer

You're familiar with using Director's score to combine images, sound, transitions, palettes, and tempos to create movies. Each time the playback head enters a frame, Director checks the score to determine the characteristics assigned to each element—sprites, sounds, transitions, palettes, and tempos—used in that frame. By making a score channel a puppet, you can tell Director to ignore the score's settings for items in that channel and change the channel's contents directly from Lingo. In a sense, the channel is a puppet and Lingo pulls the strings.

When using the score, you have to plan for and implement segments that allow for all possible combinations of events and conditions in the movie. Puppets can use fewer frames to achieve effects that could otherwise require many frames to achieve. For example, when the user clicks the mouse, you could change what is on the stage by assigning a sprite a different cast member without leaving the frame. Or you could change the palette or play a sound other than one assigned in the score in response to a user choice or some other movie event. By making these changes from Lingo, you don't need to add segments in the score that provide for all possible combinations of animation, palette change, or sound.

You are already familiar with making a sprite into a puppet from working with the section "Introducing events, messages, and handlers" in Chapter 3. The simple handler you wrote was able to change a sprite's foreground color because you first made channel 1 a puppet by using the statement `puppetSprite 1, TRUE`.

The next section, "Creating a sample puppet," is a simple exercise that demonstrates how puppets let you switch the cast member assigned to a sprite. Sections later in this chapter give you details about the specific things you can do with puppet sprites, sounds, transitions, palettes, and tempos.

Creating a sample puppet

You can make the contents of the tempo, palette, transition, sound, and sprite channels into puppets by using the appropriate Lingo commands. The following exercise shows you how to make the contents of a sprite channel into a puppet and then change the cast member that is assigned to the sprite channel using Lingo.

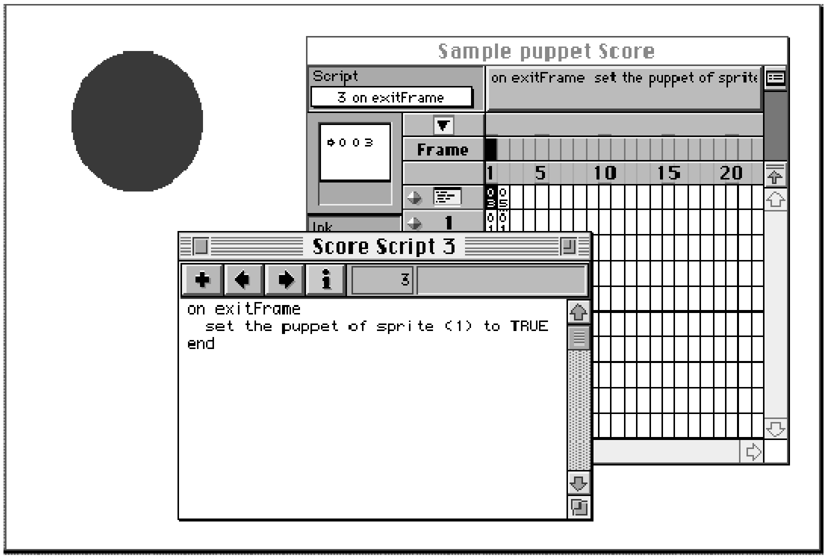
To create a puppet:

1. Create a new movie.
2. Use the tools in the paint window to create two bitmap cast members numbered cast member 1 and cast member 2.
   Use a different shape and color for each bitmap.
3. Place cast member 1 in frames 1 and 2 of channel 1.
4. Select the script channel in frame 1 of the score.
5. Click the script preview button at the top of the score.
   A new score script window appears.

6. Type the following handler:

```
on enterFrame
  set the puppet of sprite (1) to TRUE
end
```

When the playback head enters frame 1, this handler makes the contents of channel 1 a puppet.



```
on exitFrame
  set the puppet of sprite (1) to TRUE
end
```

7. Press Enter or click the close box in the script window to enter the script.

You just created a script that makes the sprite in channel 1 a puppet. Because sprite 1 is now a puppet, you can control the sprite directly from Lingo. In the next procedure, you'll write a script that uses Lingo to switch the cast member assigned to the sprite. If the sprite was not a puppet, this script would have no effect.

To switch the cast member assigned to the sprite:

1. Select channel 1 in frame 2 of the score.
2. Choose New from the Script pop-up menu.
   A new Script pop-up menu appears.
3. Write the following handler in the script window:

   ```
   on mouseUp
     set the castNum of sprite (1) to 2
   end
   ```

4. Press Enter or click the close box to enter the script.
5. Make the movie loop in frame 2 by attaching the following script to the script channel in frame 2.

   ```
   on exitFrame
     go to the frame
   end
   ```

6. Rewind and play the movie.
7. Click the sprite.
   The sprite changes to the other cast member.

You just created a movie in which you put the sprite in channel 1 under direct control of Lingo by making it a puppet. Because the sprite was a puppet, you were able to use Lingo to switch the cast member assigned to the sprite. Switching a sprite's cast member is just one thing you can do with puppet sprites. Examples of other uses for puppet sprites are included in the description of Lingo features throughout this guide.

Using puppets

As you saw in the sample movie you just created, you let Lingo control a channel by making it a puppet:

- For sprites and sounds, the channel remains a puppet until you use a statement to return control to the score. The statement you use depends on whether the channel is a sprite channel or a sound channel. See “Using puppet sprites” and “Using puppet sounds” later in this chapter for more information about turning off puppet sprites and puppet sounds.
- For tempos and palettes, the puppet condition lasts until the playback head enters a frame that has a new palette or tempo setting. For transitions, the puppet condition applies only to the specific instance in which the puppet transition is used.

The specific statements used to create and undo a puppet and what attributes of a puppet you can control are different for different types of puppets. The following sections explain the statements to use and some of the attributes you can control for different types of puppets.

Using puppet sprites

When a sprite is a puppet, you can use Lingo to control any sprite property that you can control from the score and some additional properties. This includes controlling:

- The cast member assigned to the sprite. This lets you switch the cast members in response to conditions or create animation by switching a series of cast members while the playback head stays within one frame.
- The location of a sprite. With this ability, you can control whether a sprite is draggable or reassign it to a new location. Although you can make a sprite draggable by selecting the Moveable checkbox in the score, controlling the sprite from Lingo lets you change whether it is draggable in response to movie conditions.

- The size and shape of a sprite. This lets you adjust how individual sprites move and change shape in response to other conditions in the movie.
- The color, ink, line thickness, and pattern assigned to a sprite. With Lingo, you can modify any of these conditions to fit the needs of your movie.
- The ability to edit text sprites. Although you can make sprites editable by selecting the Edit Text checkbox in the score, controlling this from Lingo lets you turn editability on and off as movie conditions require.
- The height, width, font, and style of text.

You can specify whether a sprite in a channel is a puppet by using the `puppetSprite` command, followed by the channel number, a comma, and the value `TRUE` or `FALSE`:

- `TRUE` makes the sprite in the specified channel a puppet.

  For example, the statement `puppetSprite 9, TRUE` makes the sprite in channel 9 a puppet. The statement `puppetSprite 35, TRUE` makes the sprite in channel 35 a puppet.

- `FALSE` removes the puppet condition from the sprite in the specified channel.

  For example, the statement `puppetSprite 9, FALSE` removes the puppet condition from the sprite in channel 9. The statement `puppetSprite 35, FALSE` removes the puppet condition from the sprite in channel 35.

You can also turn puppet sprites on and off by setting the `puppet of sprite` property to `TRUE` or `FALSE`:

- Declaring a sprite's `puppet of sprite` property to be `TRUE` makes the sprite a puppet.
- Declaring a sprite's `puppet of sprite` property to be `FALSE` undoes the sprite's puppet condition.

For example, the following statement makes the sprite in channel 9 a puppet by setting the puppet of sprite property to `TRUE`:

```
set the puppet of sprite 9 to TRUE
```

It is important to give control back to the score when it's no longer necessary for the channel to be a puppet. If you don't turn off a puppet when you are finished with it, unexpected results can occur when you try to control that channel from the score later on.

A puppet's initial properties are the same properties that the sprite has when it is made a puppet. The puppet's initial location, color, cast member, and other properties are the location, color, cast member, and other properties of the sprite. Subsequently, you can use other scripts to change these properties.

▶ **Tip** *The sprite channel must contain a sprite before you declare it a puppet. If you declare a channel a puppet when the channel contains no sprite, the results can be unpredictable.*

When to make a sprite a puppet

Making the sprite channel a puppet is required for controlling sprite properties from Lingo. Conversely, handlers that try to change any sprite property have no effect unless the channel is a puppet.

▶ **Tip** *In some cases, changes can be made to a sprite's properties when the playback head enters a new frame. This can seem to indicate that it isn't necessary to make a channel a puppet before changing the property. To avoid confusion, always declare sprite channels puppets before trying to change sprite properties from Lingo.*

The following sprite properties require the channel to be a puppet:

backColor	locH
blend	locV
castNum	moveableSprite
constraint	pattern
cursor	scriptNum
editableText	spriteBox
foreColor	stretch
height	trails
immediate	top
ink	visible
lineSize	width

Using puppet sounds

Puppet sounds let you use Lingo to override the sound channels of the score. Through Lingo you can:

- Play a puppet sound by using the `puppetSound` command followed by the name of the sound cast member you want to play. For example, assigning the handler:

```
on mouseUp
  puppetSound "Crickets"
  go to "Scene 4"
end
```

  to a button has the movie play the sound "crickets" whenever the button is clicked. This could be useful as part of the transition when the movie goes to a new scene.

- Turn off a sound by using the `puppetSound` command followed by zero in the form `puppetSound 0`.

As with puppet sprites, you must turn off the puppet sound condition when it is no longer appropriate. If you don't turn the puppet condition off, any subsequent sounds in the score won't play. To turn the puppet sound off and give control back to the sound channels of the score, use the statement `puppetSound 0`.

You do not need to make a sound channel a puppet to use the `sound fadeIn`, `sound fadeOut`, or `sound playFile` command or to set `the soundLevel`.

Using puppet tempos

Puppet tempos let you use Lingo to set the tempo. This is useful when you want to change the tempo of the movie in response to different conditions such as a user's action or the type of computer the movie is playing on.

You make the tempo channel a puppet by using the `puppetTempo` command followed by the new tempo, in frames per second. You can specify any integer from 1 to 60.

For example, the statement `puppetTempo 30` sets the tempo to 30 frames per second. The statement `puppetTempo vTempo` sets the tempo to the value assigned to the variable vTempo.

Using puppet transitions

Puppet transitions let you control transitions between frames. For example, you could use puppet transitions to specify one of several transitions depending on which sprites are on the stage when the playback head enters a new frame. Or you could apply a transition only to a new sprite when it appears on the stage.

You create a puppet transition by using the `puppetTransition` command followed by values that specify the nature of the transition. The parts of the statement are as follows:

`puppetTransition` *transition-type* [,*time*] [,*chunk-size*] [,*change-area*]

The parameters in this statement let you specify the same options that are in the Set Transition dialog box. When you write the actual statement:

- Always put the `puppetTransition` command first.
- Replace *transition-type* with a code number that specifies the type of transition.
- Replace the optional parameter *time* with a value for the length of time you want the transition to take.
- Replace the optional parameter *chunk-size* with the desired chunk size (number of pixels that change in each step of the transition).
- Replace the optional parameter *change-area* with `TRUE` to make the transition occur only in the areas of the stage that change during the transition. Replace *change-area* with `FALSE` to apply the transition to the entire stage. If you don't include this parameter, the transition occurs in the changing area only.

How to use the `puppetTransition` command is described in detail in the *Lingo Dictionary*.

▶ ***Tip*** *Use the updateStage command after the puppetTransition command to have the transition appear on the stage.*

The `puppetTransition` command applies only to the frame in which you issue the command. You do not need to turn off the puppet condition for the transition channel after the transition occurs.

Using puppet palettes

Puppet palettes let you use Lingo to change the current palette. This is useful when you want to change the palette to suit changing conditions in the movie without entering a new frame. For example, you could use a puppet palette to change the palette if you switch a cast member assigned to a sprite as you did in the section, "Creating a sample puppet'" earlier in this chapter.

You set a new palette by using the `puppetPalette` command followed by the cast member name or number of the new palette. If you want, you can also specify how fast or over how many frames the new palette fades in.

The form of the `puppetPalette` command is

`puppetPalette` *palette-name* `[,`*speed*`] [,`*number-of-frames*`]`

The parameters in this statement let you specify the same options that are in the Set Palette dialog box. When you write the actual statement:

- Replace *palette-name* with the name of the palette.
- Replace the optional parameter *speed* with a number from 1 to 60, with 60 being fastest, to set the speed of the fade in.
- Replace the optional parameter *number-of-frames* with the number of frames over which the palette is to fade in.

For example, the statement `puppetPalette "Rainbow" 15, 10` makes the rainbow palette the current palette. The transition to the new palette occurs over ten frames at a relative speed of 15.

When the puppet palette condition is still in effect, any subsequent palette changes in the score are ignored. As with puppet sprites, you must turn off the puppet palette condition when it is no longer appropriate.

Palette effects work fully only in 256 colors. You do not see palette transitions in thousands or millions of colors.

To turn the puppet palette off and give control back to the palette channel of the score, use the statement `puppetPalette 0`.

Chapter 5

Manipulating Sprites

Lingo gives you several ways to manipulate sprites while the movie plays. This chapter describes how to utilize these features in your movie. It tells you how to:

- Make sprites draggable
- Check a sprite's location
- Compare the locations of sprites
- Move a sprite to a new location by resetting its coordinates
- Constrain a moveable sprite
- Switch which cast member is assigned to a sprite.

Making sprites draggable

You might be familiar with using the Moveable option in the score to make sprites draggable. However, setting the `moveable of sprite` property, you can make sprites moveable independent of the score.

You make a sprite moveable by setting the sprite's `moveable of sprite` property to `TRUE` in the sprite script for the sprite or a frame script.

For example, the following handler, used in a frame script, makes sprite 1 moveable when the user clicks the sprite:

```
on enterFrame
  set the puppet of sprite 1 to TRUE
  set the moveableSprite of sprite 1 to TRUE
end
```

The `moveable of sprite` property applies only in the frame in which the command is used. The command has no effect after the playback head enters another frame or loops and re-enters the current frame. Therefore, to make the sprite unmoveable again, use the `go to` or `play` commands to enter a new frame that doesn't set the puppet's `moveableSprite of sprite` property to `TRUE`.

By setting `moveable of sprite` to `TRUE` in an if–then structure, you can make the sprite moveable in response to some user action.

If setting the `moveable of sprite` property doesn't work properly in a script, make sure:

- The statement is spelled properly
- The statement is placed in the sprite script for the frame in which you want the sprite to be moveable.

Checking a sprite's location

The ability to check, change, or restrict a sprite's location and to evaluate the cursor's place on stage can have many uses in your movies. With Lingo you can:

- ◆ Determine the exact position of a sprite or the cursor on the stage
- ◆ Determine the coordinates of locations on the stage so that other handlers can use them
- ◆ Check whether sprites overlap or how close together they are.

Combined with other Lingo, these features are the basis of much of Director's interactivity. This section explains how Lingo performs these basic tasks. For more ideas about how you can use these features in movies, see the sample movies.

Checking cursor and sprite locations

You can determine the location of the cursor or a sprite from Lingo by using:

- ◆ The elements `mouseH` and `mouseV` to measure the cursor's horizontal and vertical position
- ◆ The elements `locH of sprite` and `locV of sprite` to measure the horizontal and vertical position of a sprite's registration point.

You can evaluate the cursor location using the statements `put the mouseH` and `put the mouseV`.

To create a sample script that checks the cursor's location:

1. Create a new movie.
2. Open the score.
3. Select the script channel in frame 1, and click the script preview button.
   A new score script window appears.

4. Type the following handler in the score script window:

```
on exitFrame
  put the mouseH
  put the mouseV
  go to the frame
end
```

5. Press Enter or click the close box to enter the script.

To see the effect of the script:

1. Play the movie.
2. Open the message window.
3. Move the cursor to different places on the stage.

Each time the playback head exits the frame, the message window displays the number of pixels that the cursor is from the upper left corner of the screen:

- For `mouseV`, the value is the number of pixels the cursor is below the upper left corner of the stage. Negative values are the number of pixels above the top of the stage.
- For `mouseH`, the value is the number of pixels the cursor is to the right of the upper left corner of the stage. Negative values are the number of pixels to the left of the stage.

Similarly, you can evaluate a sprite's location by using `locV of sprite` and `locH of sprite`.

To create a sample handler that evaluates a sprite's location:

1. Create a new movie.
2. Open the message window and turn on the Trace checkbox.
3. Select frame 1 in channel 1 of the score.
4. Click the Moveable checkbox in the score window.
5. Use a tool from the tools window to draw a shape on stage.

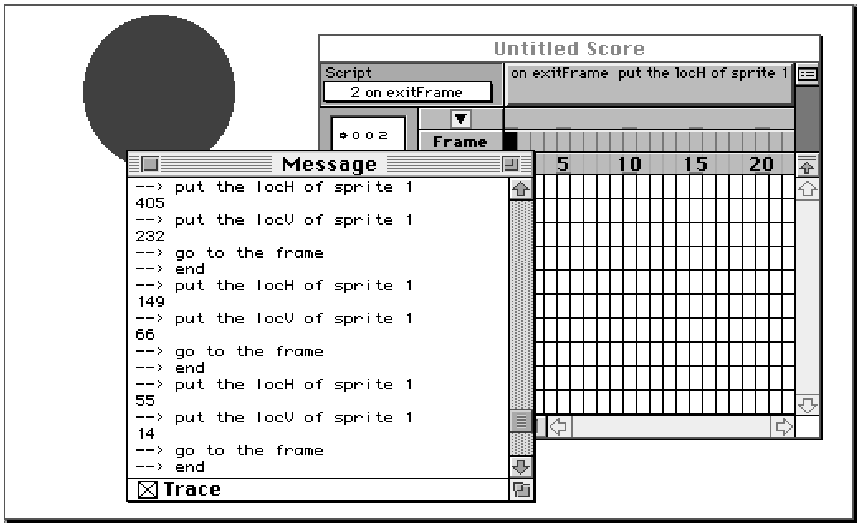
6. Include the following handler in the frame script for frame 1:

```
on exitFrame
  put the locH of sprite 1
  put the locV of sprite 1
  go to the frame
end
```

This script has the message window display the sprite's horizontal and vertical location and then has the playback head re-enter frame 1.

7. Play the movie and drag the sprite on the stage.
   You don't need to save the movie after you are done.

As you drag the sprite, the message window displays the new values for the sprite's location. This is a typical message window display that could result from dragging the sprite:



```
--> put the locH of sprite 1
405
--> put the locV of sprite 1
232
--> go to the frame
--> end
--> put the locH of sprite 1
149
--> put the locV of sprite 1
66
--> go to the frame
--> end
--> put the locH of sprite 1
55
--> put the locV of sprite 1
14
--> go to the frame
--> end
```

The elements `locH` and `locV` tell you the position of the sprite's registration point. Similarly, you can refer to the location of the sprite's bounding rectangle by using the following elements:

The element	Refers to
`the left of sprite`	The left edge of the sprite's bounding box
`the right of sprite`	The right edge of the sprite's bounding box
`the top of sprite`	The top of the sprite's bounding box
`the bottom of sprite`	The bottom of the sprite's bounding box

For example, the statement `put the top of sprite 1` displays the number of pixels from the top of sprite 1's bounding rectangle to the top of the screen.

Comparing sprite locations

You can determine whether a sprite intersects another sprite or is contained within it. For example, you might want to check for this when the user drags a sprite to a certain location as part of assembling pieces of a puzzle.

The element `sprite...intersects` tells you whether the bounding rectangle of one sprite touches the bounding rectangle of another sprite. The proper syntax for this element is as follows:

`sprite` *sprite1* `intersects` *sprite2*

When you use this element, replace *sprite1* with the channel number of one sprite you want to compare and *sprite2* with the channel number of the other sprite. The order in which you put the sprite numbers doesn't matter. The phrase `sprite 1 intersects 2` has the same effect as `sprite 2 intersects 1`.

To see how to use `intersect`, first create a movie that contains two simple sprites:

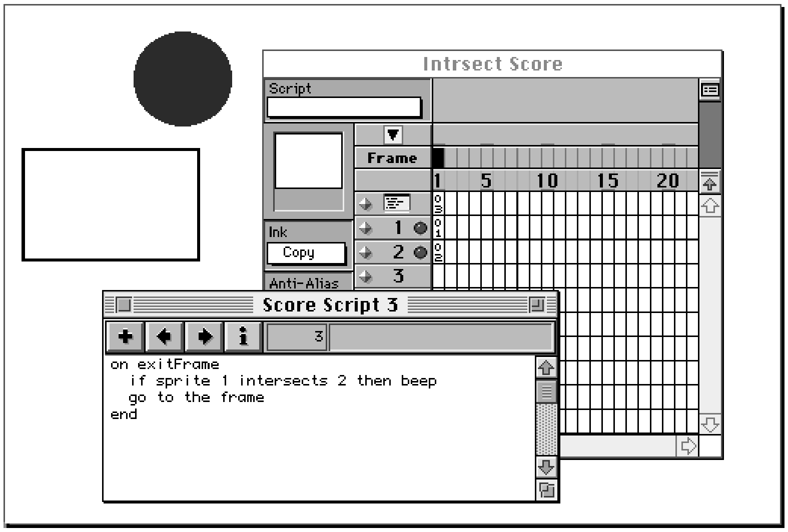
1. Choose New from the File menu.
   A new movie opens.
2. Select frame 1 in channel 1 of the score.
3. Open the tools window.
4. Draw a QuickDraw rectangle on the stage.
5. Click the Moveable checkbox in the score.
6. Select frame 1 in channel 2 of the score.
7. Draw a QuickDraw circle on the stage.
8. Click the Moveable checkbox in the score.

When you are finished, you should have a one-frame movie that contains a rectangle and a circle.

Now, write a handler that checks whether the two sprites are in contact:

1. Select the script channel for frame 1.
2. Enter the following handler into the frame script:

```
on exitFrame
  if sprite 1 intersects 2 then beep
  go to the frame
end
```



```
on exitFrame
  if sprite 1 intersects 2 then beep
  go to the frame
end
```

3. Rewind and play the movie.
4. Drag the moveable rectangle over the circle.
   The computer beeps when the bounding rectangle of one sprite intersects the other. Notice that the computer beeps even when you drag the rectangle over a corner of the circle but not the circle itself.
5. Press Command-S to save the movie and name it "Intrsect" when you are done.
   You will use this movie again in the section "Constraining a moveable sprite," later in this chapter.

Controlling sprite locations

The score normally assigns sprite locations on the stage. Lingo lets you change sprite location properties while the movie plays. This section shows you how to relocate a sprite and how to constrain a sprite that is moveable.

Changing a sprite's location

Using the `set` command, you can change a sprite's location. Remember that since you want Lingo to override the score, the sprite must be a puppet before Lingo can change its location.

To put a sprite at a specific location, set the vertical and horizontal location of the sprite to the location you want. You can specify coordinates on the screen or the position of some other sprite.

For example, this handler sets the vertical location of sprite 1 to 250 and its horizontal location to 300 when the movie enters a new frame:

```
on enterFrame
  puppetSprite 1, TRUE
  set the locV of sprite 1 to 250
  set the locH of sprite 1 to 300
  updateStage
end
```

Attaching this handler to the script for sprite 12 moves sprite 12 to the same location as sprite 8 when sprite 12 is clicked:

```
on mouseUp
  puppetSprite 12, TRUE
  set the locV of sprite 12 to the locV of sprite 8
  set the locH of sprite 12 to the locH of sprite 8
  updateStage
end
```

To locate a sprite where a mouse click occurs, set the sprite's location to the location of the mouse click. For example, attaching this handler to a frame script moves sprite 9 to the location of the mouse click:

```
on mouseDown
  puppetSprite 9, TRUE
  set the locV of sprite 9 to the mouseV
  set the locH of sprite 9 to the mouseH
  updateStage
end
```

Place this handler in a frame script because you want to detect that the mouse click anywhere on the stage, not just on specific sprites.

Constraining a moveable sprite

Sometimes you want a sprite to be moveable but restrict it to a certain region. For example, you could let someone drag furniture within the floorplan of a house but restrict motion to stay within the walls, or you could create a draggable slider with an indicator that moves across a gauge.

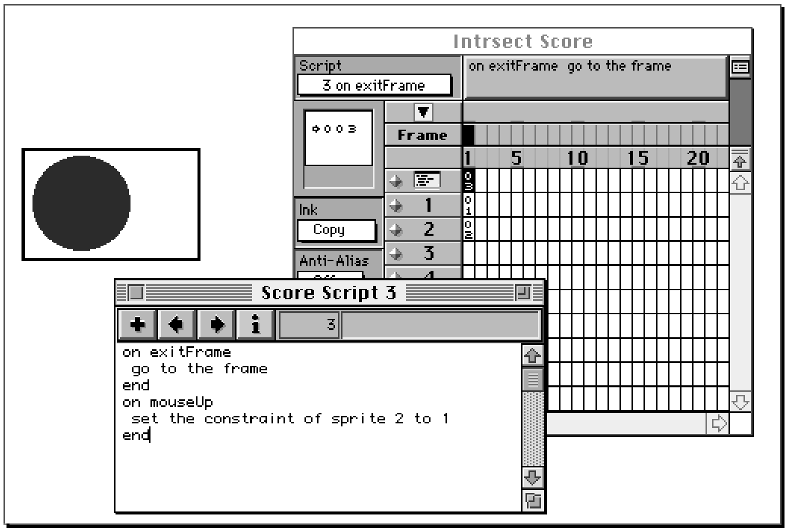
Using `the constraint of sprite` property, you can restrict a moveable sprite to stay within the area of a second sprite.

To write a sample handler that constrains a sprite to the area of another sprite:

1. Open the movie "Intrsect" that you created in the section "Comparing sprite locations," earlier in this chapter.
2. Open the script window for the frame script assigned to frame 1.
   The script contains the handler you wrote in the section "Comparing sprite locations." In this exercise, you replace that handler with a handler that constrains the circle (sprite 2) to the area of the rectangle (sprite 1).

3. Replace the frame script's on exitFrame handler with the following handlers:

```
on exitFrame
  go to the frame
end
on mouseDown
  set the constraint of sprite 2 to 1
end
```



```
on exitFrame
 go to the frame
end
on mouseUp
 set the constraint of sprite 2 to 1
end
```

4. Rewind and play the movie.

5. Drag the circle over the rectangle.

When you first click the circle, it jumps to the nearest part of the rectangle. As you drag the circle, the registration point of the draggable sprite stays within the bounding rectangle of the other sprite. If the circle doesn't move, make sure the Moveable checkbox is selected for the circle sprite. (For any QuickDraw cast member, the registration point is the upper left corner of the cast member's bounding rectangle.)

Switching a cast member assigned to a sprite

Lingo can switch the cast member that is assigned to a sprite while the movie plays. This lets you animate a sprite by cycling through a series of cast members and lets you use far fewer frames to change objects on the stage as movie conditions change. Without sprites, you would need many frames connected by `go to` and `play` commands to allow for all the possible combinations of sprites in a movie.

To write a handler that switches cast members in a movie:

1. Make the sprite a puppet using the puppetSprite command.
2. Use the phrase `set the castNum of sprite`, followed by the sprite's channel number, the term `to` or =, and the new cast member's name or number.

For example, the following handler switches sprite 2 to cast member 15 when the user clicks sprite 2:

```
on mouseDown
  puppetSprite 2, TRUE
  set the castNum of sprite 2 to 15
  updateStage
end
```

Alternatively, you could refer to the cast member by name. If a cast member had the name "Night Sky" you could use a handler similar to the following:

```
on mouseDown
  puppetSprite 2, TRUE
  set the castNum of sprite 2 to ¬
  the number of cast "Night Sky"
  updateStage
end
```

You can create animation from Lingo by using a series of such statements in succession to switch a series of cast members. For example, you could reverse a sprite's image when it is clicked by switching the sprite's cast member. For sprite number 1, if the original cast member is number 1 and the reversed image is number 2, the following handler makes the image appear to reverse on a `mouseDown` and then returns to the original image when the `mouseDown` is over:

```
on mouseDown
  set the castNum of sprite 1 to 2
  updateStage
  repeat while the stillDown
    nothing
  end repeat
  set the castNum of sprite 1 to 1
  updateStage
end
```

Chapter 6

Using the Keyboard & Mouse

Lingo lets you create movies that use the keyboard and mouse to interact with the user. This chapter shows how, by placing the right scripts in the right places, you can:

- Make text on the stage editable or noneditable
- Check which keys are pressed
- Check and combine text strings
- Modify text
- Detect when the cursor rolls over a sprite
- Check for timeouts.

Editing text

Director lets you create text that users can edit while the movie plays. This lets you use text that the user enters from the keyboard or update text by inserting text strings into a text cast member.

The score lets you make a text sprite editable in selected frames by selecting the Editable checkbox. The Text Cast Member Info dialog box lets you make a text cast member editable everywhere that the cast member is used as a sprite by selecting the Editable Text checkbox.

Lingo's `editableText of sprite` property lets you turn on or off whether text is editable as you need to during the movie. For example, suppose you want to give users 60 seconds to enter responses to a question but then make the text uneditable when time runs out. The `editableText` property lets you turn editability on and off without using the score.

To turn editable text on or off, use the statement `set the editableText of sprite`, followed by the sprite channel number, the word `to`, and then either `TRUE` or `FALSE`.

For example, to make the text sprite in channel 5 editable, use the statement `set the editableText of sprite 5 to TRUE`. To make the same text sprite uneditable use the statement `set the editableText of sprite 5 to FALSE`.

When you set the `editableText of sprite` property, place the statement in the text's sprite script. If you want the text cast member to always be editable, it is easier to do this by selecting the Editable Text checkbox in the Text Cast Info dialog box. If you want the text sprite to always be editable in specific frames of the movie, it is easier to do this by selecting the Editable checkbox in the score.

Checking keys

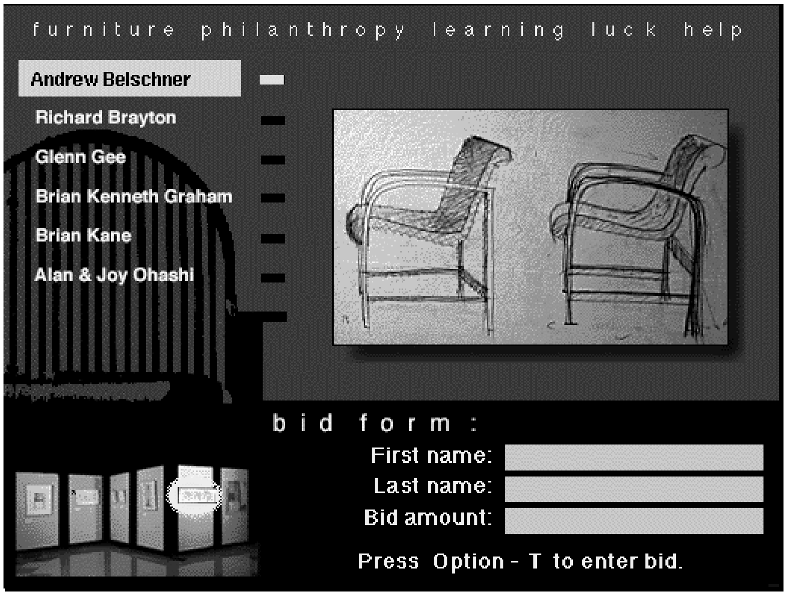
Often, you want a movie to perform a certain action when a specific key is pressed. Lingo uses the element `the key` to identify the last key that was pressed. Using `the key` as part of a script, you can check which key was pressed and specify an action in response to pressing certain keys.

Examples of checking which key is pressed appear in the bid entry section of the Luck segment of *Furniture + Philanthropy*. When users type their name and bid, what they type appears in the fields. However, pressing Return has no effect and pressing Option-T enters the bid.

In this section, you re-create the Lingo that detects when the user presses Return or Option-T and defines how the movie responds when this happens.

First, see how the finished movie responds when Return or Option-T is pressed:

1. Open and play the movie "UserKeys" in the Tutorials:Learning Lingo:Kiosk folder.
   A screen used to enter a bid appears.



First name:
Last name:
Bid amount:
Press Option - T to enter bid.

2. Type your first name in the "First name" field.

3. Press Return.

4. Type your last name.
   Your last name doesn't appear on the stage. This is because the Return was actually entered as part of the "First name" field and the field expanded out of sight. A typical user who had pressed Return at this point might expect the cursor to move to the next field.

5. Delete the text you typed in steps 2, 3, and 4, and then press Option-T.
   The character for Option-T (†) appears, because the handler that detects whether Option-T has been pressed and tells the movie to display a thank you message hasn't been written yet.

Now add the handlers that check whether the user pressed Return and filter it out if he or she did. The handler that checks for Return and Option-T will be named `whichKey`.

First, make `whichKey` the primary event handler that responds to a key being pressed:

1. Rewind the movie "UserKeys."
2. Open script cast member 1, Movie Script.
3. Scroll to the end of the on startMovie handler.
4. Insert the cursor immediately before the line `end startMovie`, and then type the following statement:

   ```
   set the keyDownscript to "whichkey"
   ```

   When you are done the last three lines of the `on startMovie` handler should look like this:

   ```
     put "" into cast "Bid display"
     set the keyDownscript to "whichkey"
   end
   ```

The statement `set the keyDown script to "whichKey"` makes `whichKey` the first handler that responds each time the user presses a key when the movie is playing.

Now, add the `whichKey` handler that checks whether Return was pressed and tells Director to ignore it if it was:

1. Scroll to the end of the movie script and type the following:

   ```
   on whichkey
     if the key = RETURN then
       dontPassEvent
     end if
   end
   ```

2. Press Enter or click the close box to enter the script.
3. Press Command-S to save your work so far.

To see the `whichKey` handler's effect on the movie:

1. Rewind and play the movie.
2. Type your first name.
3. Press Return.
   Nothing happens when you press Return; the cursor remains visible in the "First name" field.
4. Press Tab.
   The cursor advances to the "Last name" field.

Whenever the user presses a key, the phrase `if the key = RETURN` checks whether the key is the Return key. When the key is the Return key, the phrase `then dontPassEvent` prevents the message that the Return key was pressed from being passed on to anywhere else in the movie. As a result, the text cast member never knows that Return was pressed and does not include it in the text field.

You can advance to the next field by pressing Tab or clicking the field.

Next, add Lingo that has the `whichKey` handler also check whether the user presses Option-T and calls a handler that enters the user's name and bid amount into the auction.

1. Stop the movie "UserKeys."
2. Open the cast window if it is closed.
3. Double-click cast member 1, Movie Script, to open it.
4. Scroll to the end of the whichKey handler.

5. Insert the cursor between the statement `end if` and the statement `end whichKey`.

6. Add the following statements to the whichKey handler:

```
if ((the keyCode = 17) and (the optionDown)) then
  dontPassEvent
  thanksDisplay
end if
```

When you are finished, the whichKey handler should look like this:

```
on whichkey
  if the key = RETURN then
    dontPassEvent
  end if
  if ((the keyCode = 17) and ¬
  (the optionDown)) then
    dontPassEvent
    thanksDisplay
  end if
end
```

7. Press Enter or click the close box to enter the script.

The second statement
`if the (the keyCode = 17) and (the optionDown) then`
checks whether the Option and "t" keys are being pressed at the same time. When they are, the statement `dontPassEvent` prevents the fact that Option-T was pressed from passing on to anywhere else in the movie and `thanksDisplay` calls the handler `thanksDisplay`.

Next, write the handler `thanksDisplay`. The section "Modifying text fields," later in this chapter, shows you how to write Lingo that has this handler include the user' s name and bid amount in a personalized message on the screen. For now, write a handler that displays the message "Thank you for entering a bid."

To write the `thanksDisplay` handler:

1. Open cast member 1, Movie Script, if it is closed.
2. Place the cursor at the end of the script and type the following:

```
on thanksDisplay
  put "Thank you for entering a bid." into ¬
  field "Bid display"
  go to the frame + 1
end
```

3. Press Enter or click the close box to enter the script.

The `put` statement changes the content of text cast member "Bid display field" to the phrase "Thank you for entering a bid." The `go to` statement sends the playback head to frame 4, in which the text cast member "Bid display field" is on the stage.

The section "Modifying text fields," later in this chapter, explains more about how you can change the content of text cast members. For now, you only need to see the result of pressing Option-T.

To see what happens when the user presses Option-T:

1. Rewind and play the movie
2. Enter your first name, last name, and bid in the appropriate fields.
3. Press Option-T.
   The message "Thank you for entering a bid." appears.
4. Press Command-S to save your work.

The handlers you just wrote are examples of ways you specify responses that occur when certain keys are pressed. The element `the key` indicates which key was pressed.

Naturally, a common place for using `the key` is in an `on keyDown` handler. This has Lingo only check the value of `the key` when a key is actually pressed. For example, the following handler in a frame script sends the playback head to the next marker whenever the Return key is pressed:

```
on keyDown
  if the key = RETURN then go to marker (1)
end
```

You can also assign actions to more than one key. For the following handler, pressing s, h, or q gives different results:

```
on keyDown
  if the key = "s" then go to "Start"
  if the key = "h" then go to "Help"
  if the key = "q" then quit
end
```

Checking text

Lingo can check whether text contains a specific string such as a correct answer to a question, a specific name, or a letter within a set of numbers.

Three Lingo elements—`field`, `contains`, and the equals sign `(=)`—are useful for checking text. `Field` refers to the text in a text cast member. `Contains` compares two strings to see whether one string contains the other. The equals sign can determine whether a string of text is exactly the same as the contents of a text cast member. Using these lets you check whether a specified string is present in a text cast member.

For example, suppose you want people to enter "Einstein" as an answer to a question and then press Return when they are done. The following handlers offer a possible strategy for checking the answer. The first handler calls the `checkName` handler when the user presses Return. The `checkName` handler checks whether the answer is the string "Einstein." If "Einstein" is the string in the cast member named "answer," the playback head goes to the next marker. If "Einstein" is not the content of "answer," the playback head goes to the marker "Try again."

```
on keyDown
  if the key = RETURN then checkName
end
on checkName
  if field "answer" = "Einstein" then
    go to next
  else
    go to marker "Try again"
  end if
end
```

Because the first handler is activated by pressing a key and the only time you would probably want this script to run is when the movie is in this frame, it is best to put this handler in a frame script. You could put the second handler in the same script or in a movie script. Putting it in a movie script makes it easier to find the script if you want to copy it later.

Notice that the way the `checkName` handler is written, the user would need to type “Einstein” exactly. Slightly different answers—such as “Albert Einstein” or “Dr. Einstein”—would be treated as incorrect. Using `contains` to check for a string of text allows the string to contain characters in addition to the string you are checking for.

For example, the statement `if field "answer" contains "Einstein"` checks for the name “Einstein.” However, this would also correctly detect “Einstein” when the user types “Albert Einstein.” Notice, however, that `contains` could treat “Feinstein” or “Einsteiner” as correct answers.

The `contains` element also lets you search for a text string—such as a name—within text. For example, suppose the text cast member “Scientists” contains the names of noted scientists. The following handler would check for the name Maria Skoldowska and send the playback head to the marker “Radium.”

```
on testName
  if the field "Scientists" contains "Skoldowska"¬
  then go to marker "Discoveries"
end
```

Modifying text fields

As conditions change in an interactive movie, you often want to change and update text. For instance, you might want to frequently update the score of a game, people's names, or notes that the user has typed.

You can change the content of a text cast member by using the `put` commands and the `field` property. You can also combine more than one text string using the text operators `&` and `&&`:

- The `field` property specifies the text contained in a text cast member.
- The `&` operator, used between two strings, has Lingo attach the second string to the end of the first string. The `&&` operator includes a space between the two strings when they are combined.

In the section "Checking keys," earlier in this chapter, you created a handler that displayed the message "Thank you for entering a bid" when the user pressed Option-T. In this section, you replace that handler with one that puts the user's first and last name in the message.

You create this handler in the movie "UserKeys" that you worked with in the section "Checking keys." The exercise in this section requires that you have already completed the handlers in the section "Checking keys."

The movie "UserKeys" contains four text cast members: "First name," "Last name," "Bid," and "Bid display." The first three are for the first name, last name, and bid that the user enters. You use the contents of these text cast members in the text cast member "Bid display," which contains the message that appears after the user presses Option-T.

To write the handler that combines text strings:

1. Open the movie "UserKeys" that you saved from the section "Checking keys."
2. Open the cast window and double-click cast member 1, Movie Script.
3. Scroll to the end of the thanksDisplay handler.
4. Replace the statement that starts `put "Thank you ...` with the following:

   ```
   put field "First name" && ¬
   field "Last name" into fullName
   ```

   This statement uses `&&` to combine the text in "First name" and "Last name" and assigns the combined string to the variable "`fullName`."
5. On the next line, type:

   ```
   put field "Bid" into bidAmount
   ```

   This statement assigns the content of `Bid` to the variable `bidAmount.`
6. On the following line, type:

   ```
   put "Thank you," && fullName & ¬
     ", for bidding $" & bidAmount & ". Your bid has
   been entered." ¬
     into field "Bid display"
   ```

   This statement creates a new string by combining the characters you have typed here within quote marks and the strings that are assigned to the variables `fullName` and `bidAmount`.

To see the effect of the handler:

1. Rewind and play the movie.
2. Enter your first name, last name, and bid.
3. Press Option-T to enter the bid.

After you press Option-T, a message that includes the name and bid you entered appears. For example, if you entered the name Lotte Lenya and bid $500, the message would be:

“Thank you, Lotte Lenya, for bidding $500. Your bid has been entered.”

The handler you created uses the `put` command to put text into a text cast member or variable.

The handler also used the text operators `&` and `&&` to combine strings of text. Some strings were text that you typed in the statement. (These were surrounded by quotation marks.) Other strings were text in variables or cast members.

Notice that a space comes after the comma following “Thank you” because the `&&` operator inserts a space between strings. No space appears after the dollar sign (`$`) because you used the `&` operator.

Additional ways to update text fields

You can also update text by using the `set`, `put after`, `put before`, and `the text of cast` elements.

You can also use the `set` command similar to the way you used `put` to assign text to a cast member or other string, only the syntax is different. Instead of using `set` and an equal sign or the term `to`, you use `put` and the term `into`. For example, these statements do the same thing:

```
set the text of field "Greeting" to "Hello"
put "Hello" into field "Greeting"
```

You can use the term `cast` similar to the way you used `field`. For example, these statements do the same thing:

```
put "Hello" into field "Greeting"
put "Hello" into cast "Greeting"
```

Using `put...after` and `put...before`, you can insert a string of text or some other expression into a specific location within another string:

- The `put...after` command puts the string or expression at the end of the item you specify.
- The `put...before` command puts the string or expression at the beginning of the item you specify.

For example, if you used a variable named `thePrice` to record the price of an item, the statement `put "$" before thePrice` inserts a dollar sign before the price. (Note also that inserting a non-numeric character in a string of numbers, as in this example, also converts a numeric value to a string.)

You can also insert variables or other expressions. For example, if the text in text cast member number 10 is "Answer: " the statement `put 3 * 4 after field` 10 multiplies the expression 3*4 and updates the text to "Answer: 12."

The elements `char`, `item`, `line`, and `word` you identify specific characters, items, lines, or words in an expression that you can insert items before or after. For example, `word` followed by a number specifies a word in a string; `char` followed by a number specifies a character in a string. For more information about these elements, see the *Lingo Dictionary*.

For example, suppose the variable `designers` contains the designers' names "Gee Ohasi"; you can insert the name `Kaye` in the middle by using the statement `put "Kaye" before word 2 of designers`.

The result would be "Gee Kaye Ohashi."

Detecting a rollover

Many times, you want something to occur when the user rolls the cursor over a particular place on the screen. For example, you might want the cursor to change when it is over a hot spot or animate a sprite when the user passes the cursor over it.

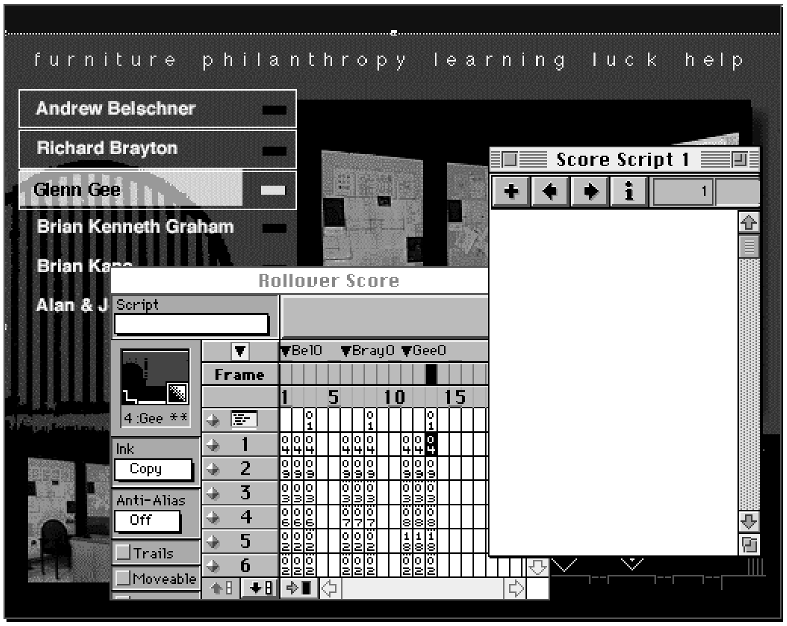
You can easily detect which sprite the cursor rolls over by using the Lingo element `rollOver` followed by the number of the sprite. This is usually done in a frame script, but it can also be done within other scripts.

The movie *Furniture + Philanthropy* uses `the rollOver` to detect when the cursor is over certain regions of the stage and changes the cursor to a magnifying glass when it is. In this exercise, you use one of the screens from *Furniture + Philanthropy* but instead use `the rollOver` to detect which designer's name the cursor is over. The names are in sprite channels 23, 24, and 25. A `go to` command then sends the playback head to a segment that displays the designer's work.

You create this handler in the movie "Rollover."

To write the handler that checks for rollovers and sends the playback head to the appropriate marker:

1. Open the movie "Rollover" in the Tutorials:Learning Lingo:Kiosk folder.
2. Open the score and notice the three markers, Bel0, Bray0, and Gee0.
   Each of the segments for these markers displays a different designer's work in the area on the right of the stage. Also, each segment has script cast member 1 assigned to it as a frame script.
3. Open the cast window and double-click cast member 1.
   The score script window opens.



4. Type the following in the score script window:

```
on enterFrame
  if rollOver (23) then go to "Bel0"
  if rollOver (24) then go to "Bray0"
  if rollOver (25) then go to "Gee0"
end

on exitframe
  go the frame
end
```

To see the effect of the handler:

1. Rewind and play the movie.
2. Move the cursor over the names Andrew Belschner, Richard Brayton, and Glenn Gee.
   The display to the right of the list of names changes when you roll over a different name.
3. Close the movie "Rollover."
   You do not need to save changes.

The `on enterFrame` handler checks whether the cursor is over sprite 23, 24, or 25 and sends the playback head to the appropriate segment if it is. The `on exitFrame` handler has the movie loop in whichever segment it is currently in.

▶ ***Tip*** *To detect a rollover in an area that is different from the bounding rectangle of a sprite, you can place an invisible QuickDraw shape over the area of the screen that you want to test for a rollover.*

Checking for timeouts

It's often important to check whether anybody has used the keyboard or the mouse for a specifed period of time. Possibly, the user is having difficulty or has just gone away. If you want, you can tell your movie what to do when the user does nothing. You do this by specifying a `timeOut` primary event handler.

For example, in the movie *Noh Tale to Tell*, the user is asked to choose a sound level from the Volume menu before the story starts. However, you could have the story start anyway after a specified time by writing Lingo that has the movie go to the start of the story when the user does nothing for the specified time.

In this section, you see how to specify a timeout handler by adding this Lingo to the movie "Timeout," which is based on the first segments of *Noh Tale to Tell.*

Specifying a `timeOut` primary event handler requires three things:

◆ Setting `the timeOutLength` to the length of time you want to pass before Director decides that a timeout has occurred

◆ Creating a handler that tells Director what to do when the timeout occurs

◆ Defining the `timeOutScript` so that it calls the handler that tells Director what to do when the timeout occurs.

To set `the timeOutLength` for the movie "Timeout":

1. Open the movie "Timeout" in the Tutorials:Learning Lingo:Storybook folder.
2. Open the score and notice that the script cells for frame 17already has a script assigned to it.
3. Open the cast window and double-click script cast member 20 to open its script window.
   Movie script window 20 appears.

4. Type the following on startMovie handler:

```
on startMovie
  set the timeoutLength to 6 * 60
end
```

   The `timeOutLength` is measured in ticks (1/60 of a second). This handler sets the length time for a timeout to 6 seconds.

   Director automatically sets the timeout length to 3 minutes unless you set it otherwise. This means that the user would have to do nothing for 3 minutes before the timeout sequence takes effect. This handler shortens the timeout to 6 seconds.

Now write the handler that tells Director what to do when the timeout occurs:

1. In the same movie script (cast member 20), add the following handler:

```
on continueWithoutClick
  go to frame "Start"
  beep
end
```

2. Press Enter or click the close box to enter the script.

This handler sends the playback head to the frame labeled “Start” whenever it is called. You can make Lingo call this handler when a timeout occurs by setting `the timeOutScript` to a calling statement for `continueWithoutClick`.

To set `the timeOutScript` to call the handler `continueWithoutClick`:

1. Select the script channel in frame 16.
2. Choose New from the Script pop-up menu.
   A new score script window appears.

3. Type the following:

   ```
   on exitFrame
     set the timeoutScript to "continueWithoutClick"
   end
   ```

   Because the movie actually loops in frame 17, setting `the timeOutScript` in an `on exitFrame` handler in frame 16 sets `the timeoutScript` the way you want before the playback head enters frame 17. The `timeOutScript` is now available when the movie is in frame 17.

4. Press Enter or click the close box to enter the script.

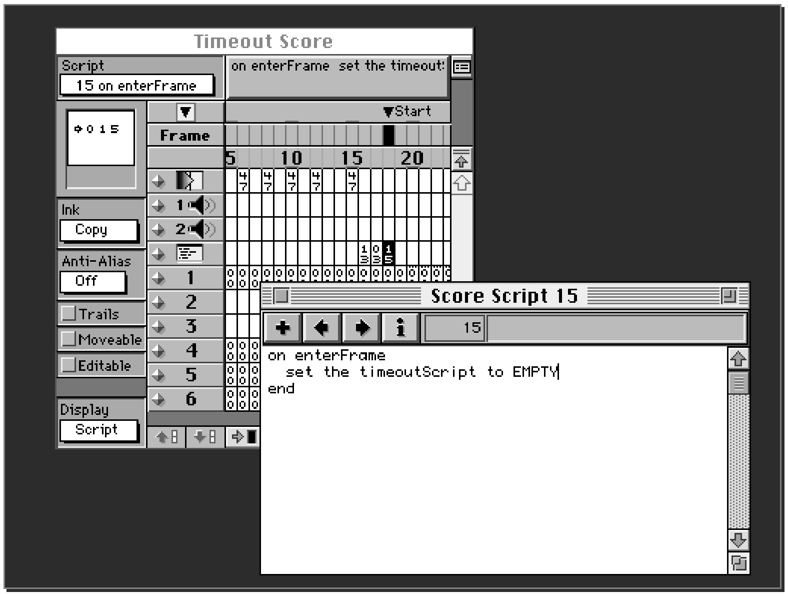
Because the mouse has a different use after the story starts, you need to reset `the timeOut Script` so that it does nothing after the story has started. Otherwise, the playback head would return to the marker "Start" every time 6 seconds passes without someone using the keyboard or mouse.

To set `the timeOutScript` to do nothing after the story starts:

1. Select the script channel in frame 18.
2. Choose New from the Script pop-up menu.
   A new score script window appears.

3. Type the following:

```
on enterFrame
  set the timeoutScript to EMPTY
end
```



```
on enterFrame
  set the timeoutScript to EMPTY
end
```

4. Press Enter or click the close box to enter the script.
5. Rewind and play the movie.

After 6 seconds at frame 17, the movie goes to the marker “Start” when you do nothing with the keyboard or mouse.

The handlers you wrote set `the timeOutScript` to perform a specific action when the user does nothing for a specified time. When the reason for the timeout script no longer applied, you removed the action by setting `the timeOutScript to EMPTY`.

Because you usually want the action that `the timeOutScript` specifies to occur in more than one frame of the movie, it is usually a good idea to define `the timeOutScript` in a movie script. When you always want the same response to a timeout everywhere in a movie, the `on startMovie` handler is a good place to specify what happens in a timeout.

Like all primary event handlers, conditions in this script are in effect unless and until you specifically deactivate it.

Chapter 7

Controlling Sound

This chapter describes the following ways that Lingo can control when and how a sound plays:

- Using puppet sounds that play when a particular event, such as a user clicking a button or a certain length of time passing, occurs when the movie plays
- Playing sound in a specific channel
- Checking whether a sound channel is currently playing a sound and making the movie respond accordingly
- Turning sound off
- Controlling sound volume from the movie instead of from the computer's volume setting and adjusting sound transitions by controlling how sound fades in and out
- Stopping a sound when a specific condition occurs in the movie

Playing puppet sounds

Puppet sounds let Lingo play sounds in response to events and conditions in addition to directions from the score. For example, putting a puppet sound statement in an `on mouseDown` handler could have a brief click sound play when the user clicks the mouse button; putting a puppet sound statement in an `on mouseUp` handler could have a sound play after the user clicks the mouse button. Or, you could have a puppet sound play when a sprite is in a specific location or a specific length of time passes.

Puppet sounds can be preloaded into RAM for faster access when the puppet sound plays. Because they can continue to play from RAM after Director leaves a movie, puppet sounds make useful transitions between movies, especially when the time required to load the next movie is noticeable.

To create a puppet sound, use the `puppetSound` command followed by the name of the sound you want to play. For example, this handler named `addThem` adds several values and plays the sound "Winner" when the total exceeds 100:

```
on addThem vFirst, vSecond, vThird
  set vTotal = vFirst + vSecond + vThird
  if vTotal >100 then
    puppetsound "winner"
    updateStage
  end if
end
```

If puppet sounds fail to play, check that there is enough RAM available to hold the sound. The sound doesn't play unless it is loaded into RAM.

To turn off a puppet sound and return control to the score's sound channels, use the statement `puppetSound 0`. For example, the following script attached to a button could make a button that turns the current sound off when the button is clicked:

```
on mouseUp
  puppetSound 0
end
```

If sounds in the score's sound channel fail to play after you use a puppet sound, make sure that the earlier puppet sound condition is turned off by using the statement `puppetSound 0`.

Playing sound in a specific channel

Lingo uses the `sound playFile` command to direct specific channels to play specific sounds. The sounds must be AIFF sounds playing from disk. Playing sounds from disk gives you the advantage of playing sounds without being limited by available RAM. However, since the computer can read only one item from disk at a time, you cannot load cast members or play more than one sound from disk when using the `sound playFile` to play sounds.

When you enter the `sound playFile` command, follow the term `sound playFile` with the channel number and sound file you want to specify.

For example, use this statement to play the sound "wind" in the first sound channel:

```
sound playFile 1, "wind"
```

Use this statement to play the sound "thunder" in the second sound channel:

```
sound playFile 2, "thunder"
```

Checking for sound conditions

You sometimes want to make actions in a movie dependent on whether a sound is playing. Using the Lingo elements `the soundEnabled` and `the soundBusy`, you can determine whether a sound is playing and have the movie respond the way you want. For example, you could display certain sprites on stage when a sound is playing or check whether a sound is playing and have animation continue until the sound finishes.

Checking sound anywhere in the movie

The `soundEnabled` property specifies whether the sound is on in Director's control panel:

- It is `TRUE` (or `1`, Lingo's numeric equivalent to `TRUE`) when sound is turned on in the control panel.
- It is `FALSE` (or `0`, Lingo's numeric equivalent to `FALSE`) when sound is turned off in the control panel.

You can test and set the `soundEnabled` property.

For example, the following statement checks whether sound is turned on or off and displays the result in the message window:

```
put the soundEnabled
```

If a sound is turned on, the message displays the number `1`, which is Lingo's numeric equivalent to `TRUE`. If sound is turned off, the message displays the number `0`, which is Lingo's numeric equivalent to `FALSE`.

This statement turns off the sound in the movie:

```
set the soundEnabled to FALSE
```

Checking sound in a specific channel

The `soundBusy` function tells whether a specific channel is playing a sound:

- It is `TRUE` when a sound is playing in the specified channel.
- It is `FALSE` when no sound is playing in the specified channel.

When you use the `soundBusy` function, follow the term `soundBusy` with the number of the sound channel you are checking.

For example, this statement checks whether sound is playing in sound channel 1:

```
put soundBusy(1)
```

This statement turns off any sound in channel 2 when there is a sound playing in channel 1:

```
if soundBusy(1) then sound stop 2
```

The `sound stop` command, which turns off the current sound in the specified sound channel, is described in the following section "Turning sound off."

▶ **Tip** *When soundBusy is used immediately after a sound starts playing, there might not be enough time for the sound to register before the soundBusy test. To avoid getting false results in this situation, use the updateStage command immediately after issuing the command that makes the sound start playing. The updateStage command also updates the speaker in this situation.*

Turning sound off

Sometimes you might want to use Lingo to turn off all sound in the movie or turn off a sound in a specific channel. For instance, when Director switches from one movie to another, you might want to turn off the sound from the first movie at a different time than the score would otherwise.

To turn off all sound in the movie, set `the soundEnabled` to `FALSE`, as explained in the previous section.

To turn off the current sound in a specific channel, use the `sound stop` command followed by the number of the sound channel that contains the sound you want to turn off.

For example, this statement turns off the current sound in channel 2:

```
sound stop 2
```

This statement turns off the sound in channel 2 when there is a sound playing in channel 1:

```
if soundBusy(1) then sound stop 2
```

Measuring a sound's time

AIFF sound files have no way to measure time. As a result, when a sound is stopped before it finishes, it has no way to tell where the sound stopped and resume play later. It can also be very difficult to synchronize sounds and animation.

By treating sounds as audio-only QuickTime movies, you can gain greater control over how the sound plays:

- The `movieTime of sprite` property can determine where a sound stops playing and re-start the sound at that or any other point in the movie.
- The `movieRate` property controls whether the sound plays slowly, fast, or in reverse.

For information about the `movieTime of sprite` and `movieRate` properties, see the entries for these elements in the *Lingo Dictionary*.

You can synchronize movie events to specific points in a sound file by issuing the `startTimer` command just before the sound starts. As the sound plays, you can use the value in `the timer` to cue other events in the movie. For example, the statement `if the timer < 180 then go to the frame` has the playback head loop in the current frame for three seconds before going on.

Controlling sound volume

Sounds played from the score's sound channels play at the volume set in the computer's sound level control. Using Lingo, you can modify the computer's sound level to suit the needs of your movie:

- Setting and testing the sound level for a specific sound channel using Lingo's `soundLevel` property.
- Fading sound in and out using Lingo's `sound fadeIn` and `sound fadeOut` commands.

Setting and testing the sound level

The `soundLevel` property refers to the computer's volume in a sound channel. Possible values are from 0 to 7, which correspond to the settings in the Macintosh Sounds control panel. Using this property, Lingo can change the sound volume directly or perform some other action when the sound is at a specified level.

The beginning of *Noh Tale to Tell* provides a Volume menu from which the user can choose the movie's volume. In this section, you see how to write the statements that set the sound level. How you can create the custom menu itself is discussed in Chapter 8, "Creating Interfaces." The sample movie you use here has the Lingo for custom menus already included.

To see where you will add statements that set the sound level:

1. Open the movie "Sound" in the Tutorials:Learning Lingo:Storybook folder.
2. Open the score and look at the segments that have markers Loud, Medium, Soft, and Mute.

The custom menu is designed to send the playback head to one of these markers, depending on which command the user chooses from the Volume menu. The frame script assigned to each frame already contains Lingo that turns checkmarks in the Volume menu on and off. For each script, you will write Lingo that sets the sound level to the level indicated by the marker label.

To add a statement that has the frame script at the marker Loud set the sound level to 7:

1. Select the script channel in frame 27.
2. Click the script preview button.
   Script window 16, which is already assigned to this cell, appears. Statements that set checkmarks for the menu have already been included in the script.
3. Insert the cursor in the first line under `on exitFrame` and type the following:

   ```
   set the soundLevel to 7
   ```

   This sets the sound level to the maximum.
4. Press Enter or click the close box to enter the script.

To add a statement that has the frame script at the marker Medium, set the sound level to medium:

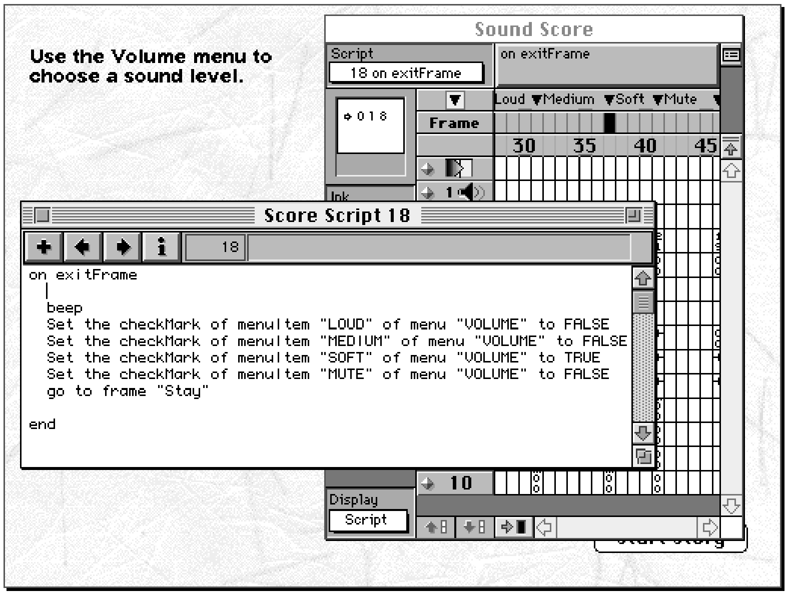
1. Select the script channel in frame 31.
2. Click the script preview button.
   Script window 17, which is already assigned to this cell, appears. Statements that set checkmarks for the menu are already in the script.
3. Insert the cursor in the first line under `on exitFrame` and type the following:

   ```
   set the soundLevel to 5
   ```

   This sets the sound level to medium.
4. Press Enter or click the close box to enter the script.

To add a statement that has the frame script at the marker Soft, set the sound level to soft:

1. Select the script channel in frame 37.
2. Click the script preview button.
   Score script window 18, which is already assigned to this cell, appears. Statements that set checkmarks for the Volume menu are already in the script.



```
on exitFrame
  |
  beep
  Set the checkMark of menuItem "LOUD" of menu "VOLUME" to FALSE
  Set the checkMark of menuItem "MEDIUM" of menu "VOLUME" to FALSE
  Set the checkMark of menuItem "SOFT" of menu "VOLUME" to TRUE
  Set the checkMark of menuItem "MUTE" of menu "VOLUME" to FALSE
  go to frame "Stay"

end
```

3. Insert the cursor in the first line under `on exitFrame` and type the following:

   `set the soundLevel to 3`

   This sets the sound level to soft.
4. Press Enter or click the close box to enter the script.

To add a statement that has the frame script at the marker Mute set the sound level to mute, which is no sound:

1. Select the script channel in frame 41.
2. Click the script preview button.
   Script window 19, which is already assigned to this cell, appears. Statements that set checkmarks for the Volume menu are already in the script.
3. Insert the cursor in the first line under `on exitFrame` and type the following:

   ```
   set the soundLevel to 0
   ```

   This sets the sound level to mute.
4. Press Enter or click the close box to enter the script.

To see the effect of the statements you added:

1. Rewind and play the movie several times.
   Each time the movie starts, choose a different command from the Volume menu. The sound plays at a different volume each time.
2. Press Command-S to save your work when you are done.

The Lingo you just wrote demonstrates a way to set the sound level. Suppose you want a particular sound, such as thunder, to play as loudly as possible but then have the computer's sound level return to its original setting when the thunder is done. You could do this with a handler similar to the following:

```
on playThunder
  set soundRecord = the soundLevel
  set the soundLevel = 7
  puppetSound "thunder"
  updateStage
  repeat while soundBusy(1)
    nothing
  end repeat
  set the soundLevel = soundRecord
end
```

This handler first records the current sound level, and then sets the sound to the maximum value of 7. When the sound is done playing in channel 1, the handler resets the sound to its original level. The repeat loop allows the sound to finish before the sound level is returned to its original value.

You could also control the sound level from a button, which gives users a way to turn sound volume up or down as the movie plays. This button handler increases the sound volume one level every time the button is clicked unless the sound level is already at 7:

```
on mouseUp
  if the soundLevel < 7 then ¬
  set the soundLevel to the soundLevel + 1
end
```

This button handler lowers the sound by one level every time the button is clicked unless the sound level is already at 0:

```
on mouseUp
  if the soundLevel > 0 then ¬
  set the soundLevel to the soundLevel - 1
end
```

Fading sound in and out

Because sounds normally play at the same volume for the duration of the sound, the beginning or end of a sound is sometimes more abrupt than you want in your movie. Lingo lets you gradually increase sound volume as the sound begins or decrease the sound volume as the sound ends. In both cases, you can specify how long the change in volume takes:

- To gradually increase sound volume as the sound begins, use the command `sound fadeIn` followed by the sound channel number.
- To gradually decrease sound volume as the sound ends, use the command `sound fadeOut` followed by the sound channel number. (Be sure to allow enough time for the sound to fade out completely before issuing another command that controls sound. Other sound-related commands might not give the expected result unless the `sound fadeOut` statement is finished.)

Both commands let you include an optional parameter that specifies the amount of time the transition takes in number of ticks (one tick is 1/60th of a second). If you don't include this parameter, the transition takes the time required to play 15 frames at the current tempo setting. (You can calculate the transition time by dividing the current tempo by 15.)

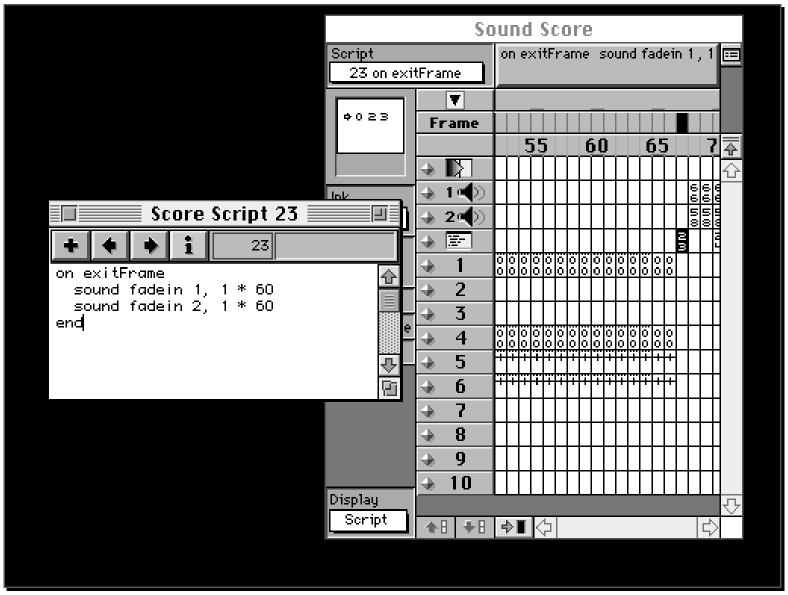
In the following exercise, you add Lingo that fades sound in and out in the movie "Sound."

To see how the sound plays without fading in and out:

1. Open the movie "Sound" that you worked with in the section, "Setting and testing the sound level."
2. Play frames 46 through 124.
   The wind and cricket sounds start and end abruptly.
3. Rewind the movie.

Now, add Lingo that fades in the cricket and wind sounds:

1. Open the score.
2. Select the script channel in frame 67, the frame before the sound begins.
3. Choose New from the Script pop-up menu.
   A new script window appears.



```
on exitFrame
  sound fadein 1, 1 * 60
  sound fadein 2, 1 * 60
end
```

4. Type the following handler:

   ```
   on exitFrame
     sound fadeIn 1, 1 * 60
     sound fadeIn 2, 1 * 60
   end
   ```

   This handler fades in the sound of the wind and crickets over 1 second.
5. Press Enter or click the close box to enter the script.

Now, add Lingo that fades out the cricket and wind sounds:

1. Open the score.
2. Select the script channel in frame 107.
3. Choose New from the Script pop-up menu.
   A new script window appears.
4. Type the following handler:

   ```
   on exitFrame
     sound fadeOut 1, 2 * 60
     sound fadeOut 2, 2 * 60
   end
   ```

   This handler fades out the sound of the wind and crickets over 2 seconds.
5. Press Enter or click the close box to enter the script.

To see the effect of the `sound fadeIn` and `sound fadeOut` commands:

1. Rewind and play the movie.
   The sound fades in starting at frame 67 and fades out starting at frame 107.
2. Stop the movie when you are finished.
   You do not need to save the changes you made.

Chapter 8

Creating Interfaces

Lingo can create familiar computer interfaces with custom menus, buttons, and cursors. This chapter tells you how to:

- Create custom menus
- Create custom cursors
- Create and set buttons.

Creating menus

Lingo can create custom pull-down menus that let the user choose from items that you provide. When the user chooses a menu item, Director runs a handler that implements the menu command.

Use the `installMenu` command to install pull-down menus. You define the actual items in the menus in a text cast member.

The Volume menu that appears in the first section of *Noh Tale to Tell* is a custom menu that lets the user choose a sound level for the movie. The volume menu disappears after the user chooses a sound level and the story begins. In this section, you will re-create the Volume menu's behavior in a movie named "MyMenus" by writing Lingo that:

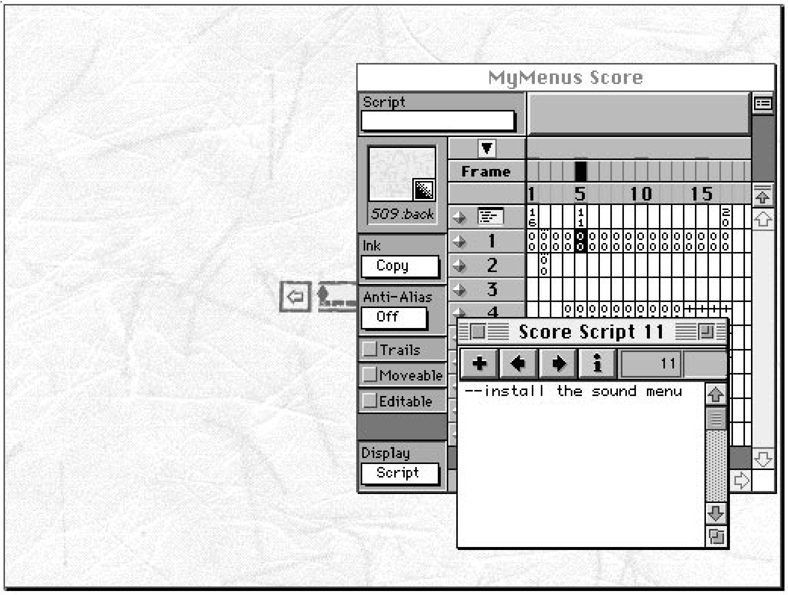
- Installs a custom Volume menu
- Defines what each menu item does when the item is selected
- Removes the Volume menu when it is no longer needed.

Installing a menu

The `installMenu` command tells Director to install the menu described in a specific text cast member. In this exercise, you write a statement that installs the menu. In the next section "Defining what's in the menu," you define the menu's contents by adding Lingo to the text cast member.

To install the Volume menu in the movie “MyMenus”:

1. Open the movie “MyMenus” in the Tutorials: Learning Lingo:Storybook folder.
2. Open the score.
3. Select the script channel in frame 5.
   Frame 5 is the first frame of the segment in which users set sound level and how long the movie pauses at the end of each scene. Script cast member 11 is already assigned to this cell and already contains the comment “`--install the sound menu`.”
4. Click the script preview button at the top of the score.
   The script window for script cast member 11 appears.



```
--install the sound menu
```

5. Type the following in the script window:

   ```
   on enterFrame
     installMenu 17
   end
   ```

   Number 17 is the number of the text cast member in which you will define the menu's contents in the next section "Defining what's in the menu."

6. Press Enter or click the close box to enter the script.
7. Press Command-S to save your work.

The statement `installMenu 17` tells Lingo to use the text in cast member number 17 for the menu's items and their definitions. You write this Lingo in the next section.

You can put menu definitions in any text cast member. Choosing cast member number 17 is just one possibility.

Note *When you specify the text cast member to use for menu item definitions, you must refer to the cast member by number—as in the statement installMenu 17—not by name.*

The menus that you create with the `installMenu` command appear at the top of the screen only after you play the movie and execute the `installMenu` command. Remember that custom menus appear only while the movie is playing.

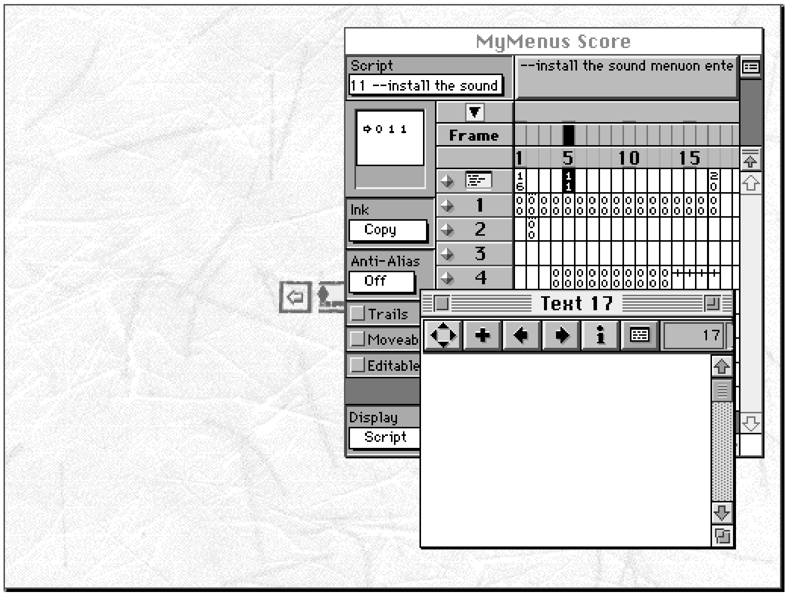
▶ ***Tip*** *If you want a menu to be available during the entire movie, define it in an on startMovie handler in a movie script.*

Defining what's in the menu

You define the actual content of a menu in a text cast member.

To define the content of the Volume menu in the movie "MyMenus":

1. Open the movie "MyMenus" if it is closed.
2. Open the cast window.
3. Double-click text cast member 17 to open it.
   Text cast member 17 is the cast member you specified earlier in the `installMenu 17` statement as the cast member that contains the definitions for the Volume menu. The text window for cast member 17 appears.



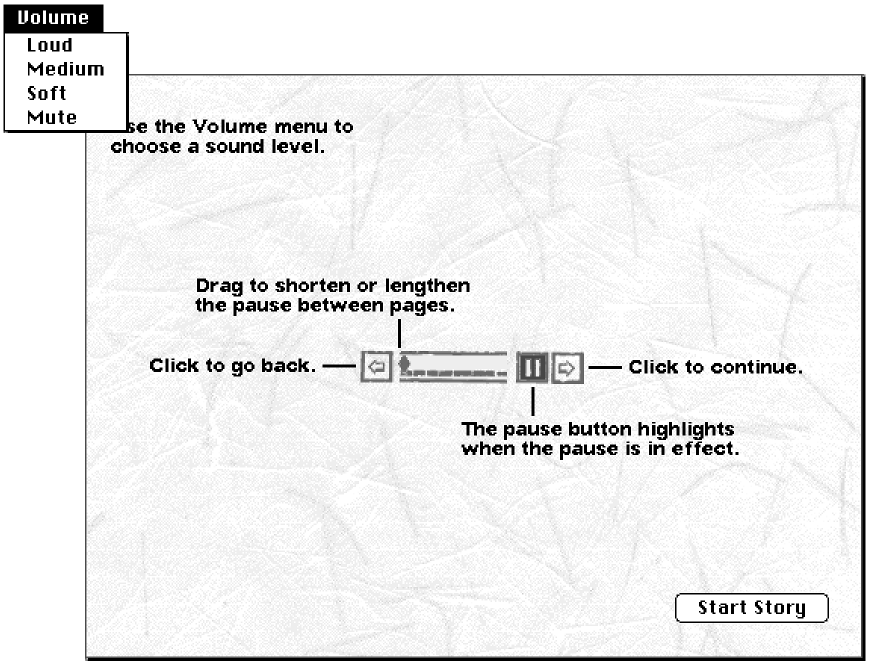
4. Type the following in the text window. (Press Option-X to create the "≈" character):

```
menu: Volume
Loud ≈ go to frame "Loud"
Medium ≈ go to frame "Medium"
Soft ≈ go to frame "Soft"
Mute ≈ go to frame "Mute"
```

5. Press Enter or click the close box to enter the script.

To see the effect of the Volume menu that you installed:

1. Rewind and play the movie.
2. When the Volume menu appears, choose different commands from the menu.
   The computer beeps each time you choose a menu item. The sound level of the beep corresponds to what you choose.


Volume
Loud
Medium
Soft
Mute
se the Volume menu to choose a sound level.
Drag to shorten or lengthen the pause between pages.
Click to go back.
Click to continue.
The pause button highlights when the pause is in effect.
Start Story

3. Stop the movie.
4. Command-S to save your work when you are done.

The `menu:` command establishes the menu's name as Volume. Text that you type on each new line after that creates a new item in the menu. The "≈" sign (created by typing Option-X on the keyboard) does not appear in the menu item. However, the statement that comes after the ≈ sign is the Lingo that runs when you choose that item from the menu. The menu item can be only one line long. You can use more than one statement for the menu item by making the line a calling statement for a separate handler.

In this example, the statements that you wrote send the playback head to the segment Loud, Medium, Soft, or Mute. The `set the soundLevel` command that has already been included in the frame script assigned to each segment is the same Lingo that you used in the section "Setting and testing the sound level," in Chapter 7 to set the sound level.

You can install additional menus by adding more sets of menu statements, either before or after existing ones, in the same text cast member. The menus appear at the top of the screen in the order that you define them.

When you append a slash (/) followed by a letter to the end of the Lingo statements that define the menu items, the letter automatically appears with the Command key symbol in your menu. Like other command key equivalents, pressing these Command keys executes the same commands as choosing the menu items themselves. For example, putting "/P" at the end of the Lingo statement makes Command-P a keyboard shortcut for the menu item and displays the Command-P symbol after the item in the menu. Note that if your custom menu uses keyboard shortcuts, your Director shortcuts don't work while the custom menu is in effect.

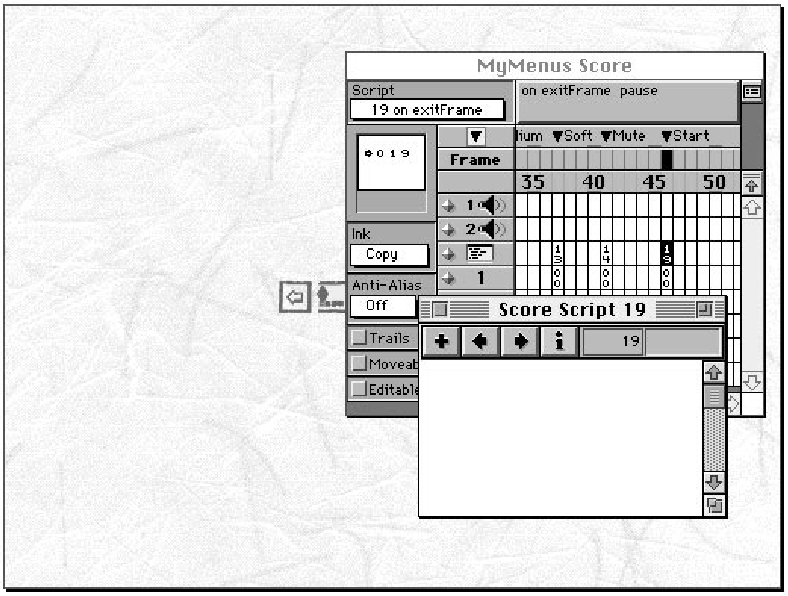
There are other special symbols that you can use in defining menus (for example, to make items bold or disabled) or placing checkmarks next to items that are selected. Refer to the `menu:` keyword and `checkMark of menuItem` property in the *Lingo Dictionary* for more information.

Removing a menu

You can remove a menu when you no longer require it by using the command `installMenu 0`. For example, in *Noh Tale to Tell*, there is no need for the Volume menu to appear after the story starts.

To remove the Volume menu from the movie "MyMenus" when the story starts:

1. Open the movie "MyMenus" if it is closed.
2. Open the score.
3. Select the script channel in frame 46.
   Frame 46 is the first frame of the actual story. Script 19 has already been assigned to the script channel in frame 46.
4. Click the script preview button at the top of the score.
   The script window appears.



5. Type the following in the script window:

```
on enterFrame
  installMenu 0
end

on exitFrame
  pause
end
```

6. Press Enter or click the close box to enter the script.

To see the effect of the script:

1. Rewind and play the movie.
2. Choose a sound level from the Volume menu.
3. Click the Start Story button.
   The Volume menu disappears after the story starts. The `pause` command pauses the movie at the end of the last frame.
4. Press Command-S to save your work when you are done.

Creating cursors

Lingo lets you replace your computer's standard cursors with custom cursors. The cursor can be any 1-bit black-and-white bitmap or combination of overlaid 1-bit black-and-white bitmaps that are at least 16X16 pixels. When the bitmap's length or width is greater than 16 pixels, Director crops the bitmap to 16X16 pixels from the upper left corner of the bitmap:

- Using the `cursor` command, you can change the cursor for the entire movie.
- Using `set the cursor of sprite` followed by a list containing the bitmap or bitmaps used for the cursor, you can have the cursor change when the cursor is over specified sprites.

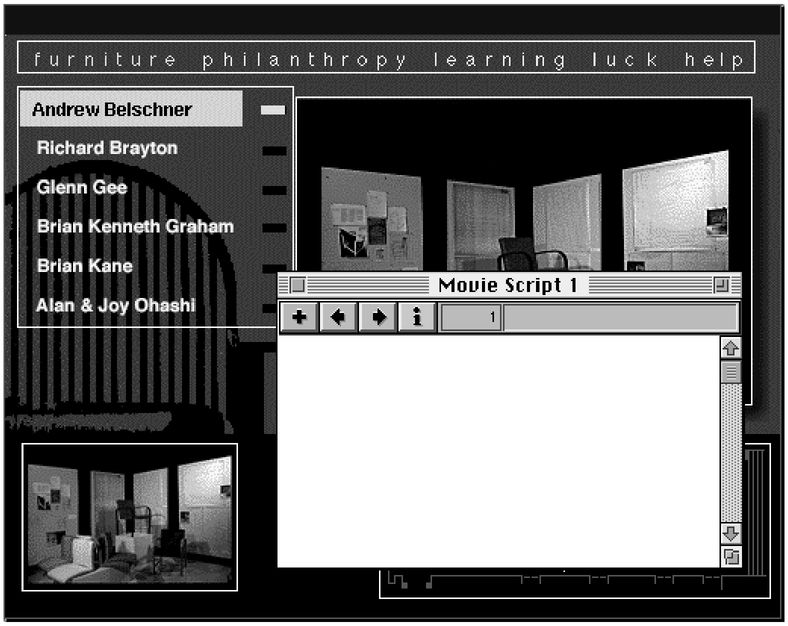
The bitmap images are available as cast members in the cast window. This is a simpler way to provide custom cursors than was used in earlier versions of Director, which treated custom cursors as external resource files.

▶ **Tip** *You can use an image that is smaller than 16X16 pixels for a custom cursor by making the bitmap's bounding rectangle at least 16X16 pixels. The invisible boundary doesn't appear on the stage.*

In *Furniture + Philanthropy*, the cursor changes to a magnifying glass when it is over certain areas of the stage. In this section, you work with a movie named "Cursors" to write Lingo that changes the cursor to a magnifying glass when it is over specific sprites. Later you'll write Lingo that changes the cursor for the entire movie.

To change the cursor to a magnifying glass when it is over certain sprites:

1. Open the movie "Cursors" in the Tutorials:Learning Lingo:Kiosk folder.
2. Open the cast window.
   The bitmap that you will use for the cursor is cast member 16.
3. Double-click cast member 1, the Movie Script.
   The movie script window appears.



4. Type the following in the Movie Script:

```
on startMovie
  set myCursor to [16]
  set the cursor of sprite 22 to myCursor
  set the cursor of sprite 23 to myCursor
  set the cursor of sprite 24 to myCursor
  set the cursor of sprite 25 to myCursor
  set the cursor of sprite 26 to myCursor
end
```

   The first statement after `on startMovie` creates the variable myCursor and sets it equal to a list that contains the bitmap used as the cursor. If you were overlaying two bitmaps to create a mask, you would list them all in the list, with the matte cast member second in the list.

   The statements that start with `set the cursor` change the cursor to the bitmaps in the list myCursor whenever the cursor is over the specified sprites. Sprites 22 to 26 are the regions that are outlined in white in the movie "Cursors."

5. Press Enter or click the close box to enter the script.

To see the effect of the script:

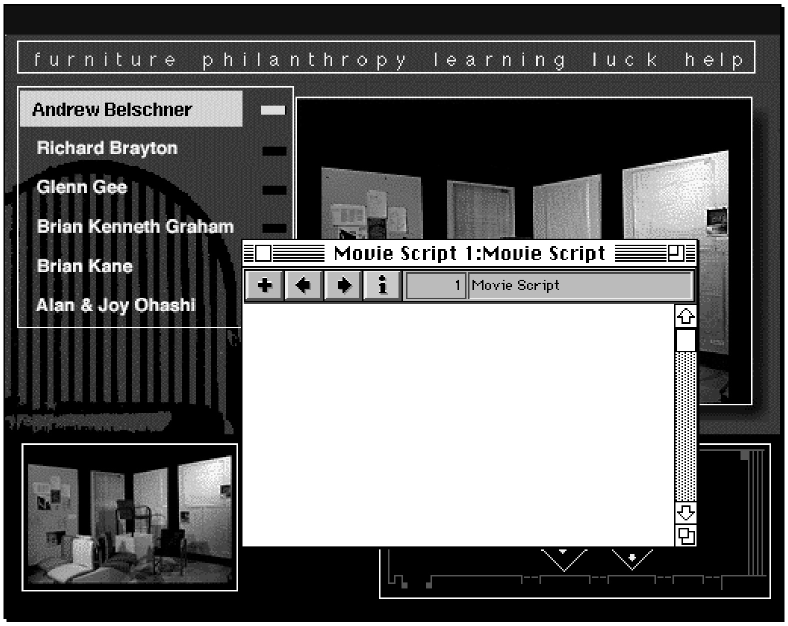
1. Rewind and play the movie.
2. Pass the cursor over different regions of the movie.
   The cursor changes to a magnifying glass when it is over the regions that are outlined in white.

You just wrote a script that changes the cursor to a custom cursor when the cursor is over sprites that you specified. You set the cursor to the bitmap contained in the list. In this case, the bitmap was cast member 16. Placing this Lingo in an `on startMovie` handler had the cursor change to a magnifying glass any time after the movie started. You could have had the cursor change only in selected frames by placing this Lingo in an `on enterFrame` or `on exitFrame` handler.

Next, you replace this handler with a handler that uses the magnifying glass everywhere on the stage by using the `cursor` command.

To use the magnifying glass as a cursor everywhere on the stage:

1. Re-open the movie "Cursors" if it is not open.
2. Open the cast window.
3. Double-click cast member 1, the Movie Script.
   The movie script window appears.
4. Delete any existing handlers that are in the movie script.



5. Type the following in the movie script window:

```
on startMovie
  set myCursor to [16]
  cursor myCursor
end
```

The first statement after `on startMovie` creates the variable `myCursor` and sets it equal to a list that contains the bitmap used as the cursor. If you were using an overlay of two bitmaps, you would put both of them in the list.

The second statement declares that the bitmap assigned to the variable `myCursor` is the cursor.

6. Press Enter or click the close box to enter the script.

To see the effect of the script:

1. Rewind and play the movie.
2. Pass the cursor over different regions of the movie.
   The magnifying glass is the cursor over the entire stage.

You just wrote a script that changes the cursor to a custom cursor. You set the cursor to the bitmap contained in the list. In this case, the bitmap was cast member 16. As when you used the `set the cursor of sprite`, placing this Lingo in an `on startMovie` handler had the cursor change to a magnifying glass any time after the movie started. You could have had the cursor change only in selected frames by placing this Lingo in an `on enterFrame` or `on exitFrame` handler.

▶ ***Tip*** *To create a mask for a cursor, make a duplicate of the cursor cast member. Use the no shrink rectangle when you cut and copy the original bitmap in the paint window. Set the duplicate cursor's ink effect to transparent.*

Creating buttons

The button, checkbox, and radio button tools in the tools window are designed to automatically behave the way that interface buttons typically work:

- A checkbox toggles between empty and selected (marked with an X)
- A radio button toggles between an empty circle and a bull's-eye
- A button highlights on `mouseDown` and returns to its original state on `mouseUp`.

You can use Lingo to react to these button clicks. In addition, you can control the properties associated with buttons.

To make a button:

1. Click the tool you want to use.
2. Drag a rectangle on the stage.
3. Type the text you want to appear on or next to the button.
4. If desired, set the font, style, size, and foreground and background color.

The button is placed in the cast as a button cast member. Add it to the movie by dragging it onto the stage.

Director automatically duplicates the standard change in a button's appearance that occurs when the button is clicked. All you need to do is provide the Lingo that tells Director what to do when the button is clicked. You can change the appearance of any checkbox or radio button (selected or deselected) through Lingo.

Controlling buttons

Lingo lets you check whether buttons are clicked and write handlers that respond accordingly and control the buttons' appearance.

Reacting to a clicked button

For a button placed on the stage over several frames, nothing happens if the user does not click the button. When the user clicks the button, the button toggles and any script associated with the button is executed:

- When the script is in an `on mouseDown` handler, the script executes when the button is pressed.
- When the script is in an `on mouseUp` handler, the script executes when the button is released.

You can put just about any script in the button, including one that calls other handlers defined in a movie script, score script, or the button's own cast script. The text associated with a checkbox can be made editable.

Reacting to a clicked checkbox or radio button

The `hilite of cast` indicates whether a checkbox or radio button is selected or deselected:

- If `the hilite of cast` is `TRUE`, the radio button or checkbox is selected.
- If `the hilite of cast` is `FALSE`, the radio button or checkbox is deselected.

In your scripts, you can test the state of the checkbox or radio button and then cause some action based on the result. In Lingo, you refer to a button by its cast name or number. For example, the following script tests whether the button cast member "Button Choice" is selected:

```
if the hilite of cast "Button Choice" = TRUE then ¬
go to frame "new sequence"
```

A button's `hilite of cast` property stores whether or not the user selected it. Each button click causes the value to switch between `TRUE` and `FALSE` (a 1 or 0). In the preceding example, if `the hilite of cast` is `TRUE` the statement sends the playback head to the frame that has the marker "`New sequence`." If `the hilite of cast` is `FALSE`, nothing happens.

Selecting a checkbox or radio button

You should not depend on the user to set the state of a checkbox or radio button. For example, if you provide a set of radio buttons, you need to make sure that the user can select only one at a time. Your script needs to establish that all the buttons except the one selected are set to `FALSE`.

To select a checkbox or radio button, you would use a `set` statement similar to this one:

```
set the hilite of cast "Button Choice" to TRUE
```

If you set the `hilite` property to `FALSE`, the checkbox or radio button is deselected.

The checkBoxAccess property

By default a user can select or deselect a checkbox or radio button. There are two more restricted types of access:

- Setting the `checkBoxAccess` property to 1 allows the user to select but not deselect the checkbox or radio button.
- Setting the `checkBoxAccess` property to 2 allows the user to neither select nor deselect the checkbox or radio button.

To change the kind of access the user has to checkboxes and radio buttons, you use the `checkBoxAccess` property. The following statement prevents the user from changing any checkbox:

```
set the checkBoxAccess to 2
```

A `checkBoxAccess` setting of 0 allows the user to select and deselect checkboxes and radio buttons. A setting of 1 allows the user to only select checkboxes and radio buttons, and prevents a selected checkbox or radio button from being deselected.

The checkBoxType property

By default a selected checkbox displays an X in the box. You can change this box to a small black square or a filled checkbox.

Use the `checkBoxType` property to change this setting:

```
set the checkBoxType to 2
```

A `checkBoxType` setting of 0 displays a standard X check. A `checkBoxType` setting of 1 displays a small square within the checkbox. A `checkBoxType` setting of 2 displays a filled checkbox.

The `checkBoxType` is a movie property. Its setting affects all displayed checkboxes.

Chapter 9

Movies in a Window

This chapter tells you about how to play fully dynamic and interactive movies in conjunction with your primary movie, by creating windows that can play movies. The material describes:

- Important background information about how to create and use lists
- How to create windows and specify the characteristics you want
- How to assign a movie to play in the window
- How to close the window when you are done.

Using lists

Lists give you an efficient way to keep track of and update an array of data. Some types of data that you might want to track in a movie include the names that users enter in a kiosk, the current windows and the movies assigned to them, the coordinates of a sprite, or the current child objects. (Child objects are described in Chapter 10, "Parent Scripts and Child Objects.")

Director offers two types of lists:

- Linear lists, in which each element is a single value
- Property lists, in which each element consists of a property and a value separated by a colon.

Both linear and property lists can be unsorted or sorted in alphabetic order.

Note *Lists are an alternative to factories and mGet and mPut. You do not have to worry about explicitly disposing of lists. Lists are automatically disposed of when they are no longer referred to by any variable.*

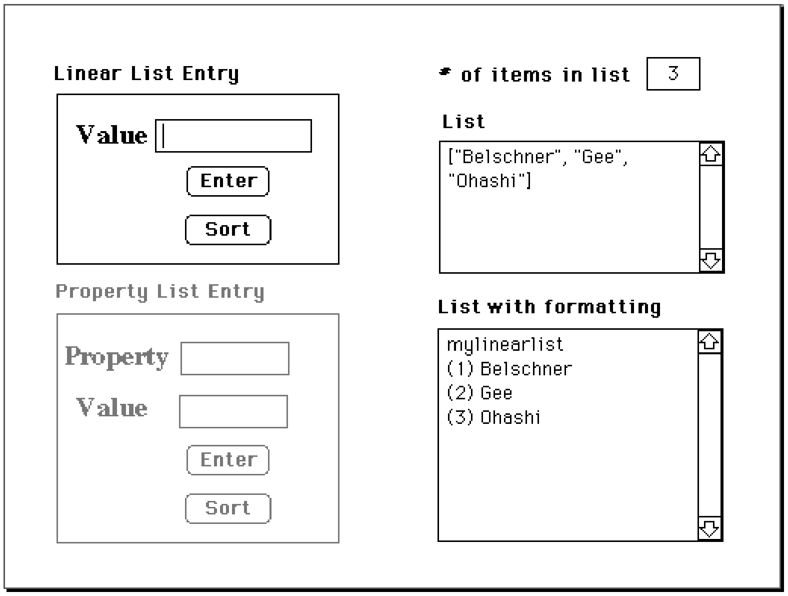
In the section "Creating cursors," in Chapter 8, you used a simple list in the statement `set myCursor to [16]`. The term `[16]` is actually a one-element list that includes the cast number of the bitmap you are using as a cursor. If you had used a second bitmap as a mask, you would have listed the cast number of the second bitmap following the first cast number. The new list would be `[16, 17]`.

Lingo can create, sort, add to, reorder, or substitute a list's contents. Before studying the Lingo that does this, look at a simple movie that uses Lingo to create and manipulate lists.

▶ ***Tip*** *In earlier versions of Director, you might have handled a set of items by making them strings in a text cast member and then referring to them by item number. Using lists is an easier approach that allows Lingo to execute faster.*

To see an example of a linear list:

1. Open and play the movie "Lists" in the Tutorials:Learning Lingo: Simulation folder.
   The movie appears. The left side contains two sets of fields where you type entries and buttons for entering and sorting. The right side contains fields that update to display the list's contents after you type entries.
2. Click the Linear List Entry section and type the name "Ohashi" in the Value field.



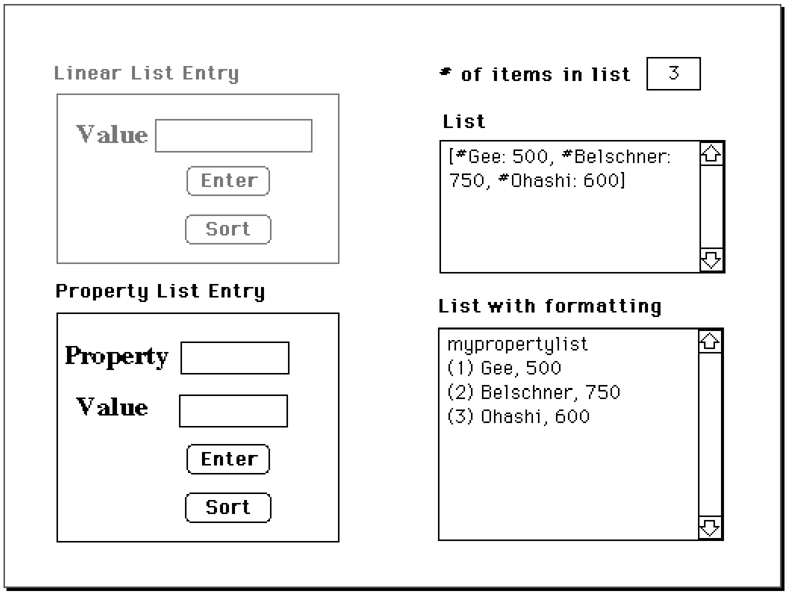
```
["Belschner", "Gee",
"Ohashi"]
```

```
mylinearlist
(1) Belschner
(2) Gee
(3) Ohashi
```

3. Click the Enter button.
   The name appears in the List and "List with formatting" fields at the right.
4. Type the name `Belschner`, and click the Enter button.
5. Type the name `Gee`, and click the Enter button.
   The names appear in the List field in the order you type them. Each element in the list is surrounded by quotation marks. The names appear in the "List with formatting" field with their list indexes.
6. Click the Sort button.
   The names now appear in alphabetical order.

To see an example of a property list:

1. Stop, rewind, and play the movie "Lists."
   The movie appears with the Value and Property fields clear.
2. Click the Property List Entry section and type the name `Ohashi` in the Property field.
3. Type a number in the Value field, and then click the Enter button.
   The name "Ohashi", followed by a colon and then the number you typed appears in the List field. The number of the position in the list, the name, a comma, and the number you typed appear in the "List with formatting" field.



```
[#Gee: 500, #Belschner:
750, #Ohashi: 600]
```

```
mypropertylist
(1) Gee, 500
(2) Belschner, 750
(3) Ohashi, 600
```

4. Type the name `Belschner` in the Property field, a number in the Value field, and then click the Enter button.
5. Type the name `Gee`, and click the Enter button.

6. Type several more property names and number values. Click the Enter button after each property and value you enter.
   The names and numbers appear in the List field in the order you type them.
7. Click the Sort button.
   The names now appear in alphabetical order in both lists.

The movie you just explored uses several Lingo commands to record and modify the lists of values and properties that you typed.

Creating lists

You specify which items are in a list by enclosing the items in square brackets. You can also use the `list()` function with the list items within the parentheses to create the list:

- Each item in a linear list consists of one value. Text strings used as an element in a linear list must be surrounded by quotation marks. Text not surrounded by quotation marks is treated as a variable. If the text has no value assigned to it, Lingo treats the item as void in the list.

  For example, the names that you entered in the linear list could be defined as list elements by using `["Ohashi", "Belschner", "Gee"]` or `list("Ohashi", "Belschner", "Gee")`.

- Each item in a property list consists of a property followed by a colon and then the value associated with the property. Properties can appear more than once in a property list. The symbol operator (#) must precede each text string that you use as a value in a property list or Lingo treats the element as an empty or void entry.

  For example, the names and numbers in the property list could be defined as list elements by using `[Ohashi:500, Belschner:750, Gee:600]`. If the numbers were properties and the names were values, you could declare the items to be list elements by using `[500:#Ohashi, 755:#Belschner, 600:#Gee]`.

You create a list by assigning the list's definition to a variable.

To create a sample linear list:

1. Type the following in the message window:

   ```
   set myList = list("b", "a", "d", "c")
   ```

2. Press Return and then type:

   ```
   put myList
   ```

3. Press Return.
   The message window displays `["b", "a", "d", "c"]`, which is the list you defined.

To create a sample property list:

1. Create a new movie.
2. Open the message window.
3. Type the following:

   ```
   set myList = [b:1, a:2, d:3, c:4]
   ```

4. Press Return and then type:

   ```
   put myList
   ```

5. Press Return.
   The message window displays `[#b:1, #a:2, #d:3, #c:4]`.

A linear list or a property list can contain no values at all. An empty linear list consists of two square brackets (`[]`). An empty property list consists of two square brackets surrounding a colon (`[:]`). Clear a linear list by setting the list to `[]`. Clear a property list by setting the list to `[:]`.

For example, the statement `set myList = [:]` clears the contents of `myList`, which is a property list.

Sorting lists

Lists can be sorted in alphanumeric order. A sorted linear list is ordered according to the values in the list. A sorted property list is ordered according to the properties in the list.

To sort a list:

▶ Use the `sort` command followed by the list's name.
For example, these statements create and then sort a list:

```
set myList = ["a", "e", "c"]
sort myList
put myList
-- ["a", "c", "e"]
```

Checking items in a list

You can determine the values at locations in a list and how many items the list contains.

To determine how many items are in a list:

▶ Use the count function.
For example, if `myList` contains three items, the statement `put count(myList)` displays the number 3 in the message window.

The following functions tell you characteristics of a list:

To check a list's	Use this function
Type	`ilk`
Maximum value	`max`
Minimum value	`min`
Value at a specific position in a list	`getAt`
Property associated with a specific position in a property list	`getPropAt`
Position of a specific property	`findPos, findPosNear, or getOne`

Adding items to a list

You can add items to the end of a list or in a specific place within the list. The commands used to add items have the following different uses:

To add an item at	Use this command
The end of a list	`append`
Its proper order in a sorted list	`add`
A specific place in a linear list	`addAt`
A specific position in a property list	`addProp`

Changing items in a list

You can replace or delete items in a list.

To delete an item from a list:

▶ Use the `deleteAt` or `deleteProp` command.

To replace an item in a list:

▶ Use the `setAt` command.

Copying lists

Assigning a list to a variable and then assigning that variable to a second variable does not make a separate copy of the list. For example, the statement `put list("Asia", "Africa") into landList` creates a list that contains the names of two continents. The statement `put landList into continentList` assigns the same list to the variable `continentList`. However, then adding "Australia" to `landList` using the statement `add landList, "Australia"` automatically adds "Australia" to `continentList` also.

You can make a copy of a list by using the `value` and `string` functions to treat the list's contents as a text string and copying the string. For example, the statement:

```
put value(string(landList)) into continentList
```

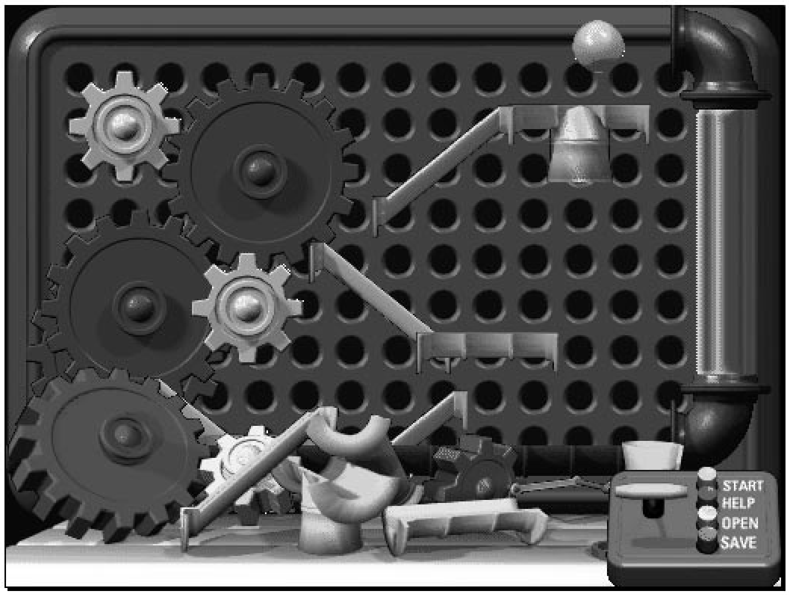
makes a copy of `landList` and assigns it to `continentList`.

What a movie in a window is

A movie in a window is a fully functional, distinct movie that can be opened by the current movie.

In one sense, the movie in a window is a child of the current movie: the current movie opens and can dispose of the window. However, each movie in a window plays like any other Director movie, with all its Lingo intact and capable of interacting with other movies. Because the parent and child movies are fully functional, movies in a window let you play two or more movies simultaneously.

The *Lingo Expo* is an example of a movie that plays other movies in a window. The interface—which you use to select and control a movie—is actually the movie named "Navigator," the parent movie. The sample movies play in a window within "Navigator." Notice that as the sample movie plays, you can interact with the sample movie the same as when it plays on its own. In this figure, the movie *MECH* is playing within the Navigator.



The typical “life” of a movie in a window involves creating the window by assigning a movie to the window; opening the window and playing the movie; and deleting the window when the reason for playing the movie no longer applies.

Any time after you assign a movie to a window, you can specify the window’s appearance—whether it is visible, has a frame and title, or is in front of or behind other windows on the screen. For convenience, you can assign these values to a variable and then refer to the variable.

You achieve each of these behaviors by using the appropriate Lingo, as described in the following sections. You see how this Lingo is implemented by re-creating it in the tutorial movie “MIAW.” The movie plays a digital video in the simulated monitor screen.

Creating a sample movie in a window

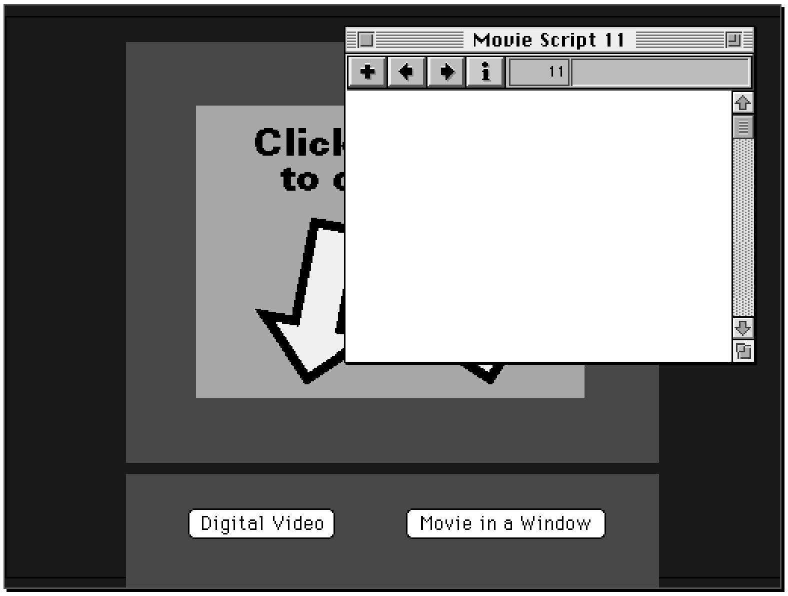
You create a window by specifying the screen rectangle for the window and then specifying the movie assigned to the window. You can also make the window visible, change its type, set its title, or set the window's size and location.

In this section, you'll write handlers for "MIAW" that create and play a movie in a window when the playback head enters frame 10. This is the frame that the movie enters when you click the Movie in a Window button.

The frame script for frame 10 already has an `on enterFrame` handler that contains a calling statement for the handler named `beginMyMovie`. You will write the `beginMyMovie` handler in a movie script.

To start writing the `beginMyMovie` handler:

1. Open the cast window.
2. Double-click cast member 11, Movie Script.
3. The movie script window appears.



To write statements that create the window:

1. Insert the cursor at top of the script window and type the following:

```
on beginMyMovie
  global myWindow
  if objectP(myWindow) then
    forget myWindow
  end if
  set horzOrigin to 127
  set vertOrigin to 66
```

The first statement declares `myWindow` a global variable. The if–then structure tests whether an object named `myWindow` already exists and deletes it if it does. (This is done to ensure that no window exists if the movie has played previously. The `forget` command is explained in the section "Closing windows," later in this chapter.) The last two statements define variables that you use to specify stage locations for the window.

2. On the following line, type:

   ```
   set myWindowRect to rect(horzorgin, ¬
   vertorgin, horzorgin + 256, vertorgin + 192)
   ```

   This statement creates a variable named `myWindowRect` that is a rectangle with the four coordinates you specified in step 1. The `rect` function defines these coordinates as a rectangle.

3. On the following line, type:

   ```
   set myWindow to window "wallMovie"
   ```

   This statement assigns the phrase `window "wallMovie"`to the variable `myWindow`. Lingo refers to a specific window using the term `window` followed by the window's name. Making the entire phrase a variable makes it easier to manipulate.

4. On the following line, type:

   ```
   set the rect of myWindow to myWindowRect
   ```

   This statement makes the rectangle described by the variable `myWindowRect` the rectangle for the window.

The Lingo you just wrote specifies the coordinates of a rectangle and then tells Director to use that rectangle as the window named `wallMovie`. By setting many of the values to variables and then using the variables, you made it easier to update the statements you are writing and to re-use them for other windows and movies by redefining variables instead of writing new handlers.

Next, tell Director which movie to play in the window, whether to make the title visible, and to open a window:

1. On the line following the statement `set the rect of myWindow to myWindowRect`, type the following:

   ```
   set the fileName of myWindow to "INTERACT"
   ```

   This statement makes the movie “INTERACT” the movie that plays in the window.

2. On the following line, type:

   ```
   set the titleVisible of myWindow to FALSE
   ```

   The `titleVisible of window` property specifies whether the window’s title bar appears. Setting it to `FALSE` makes the title bar invisible.

3. On the following line, type:

   ```
   open myWindow
   end
   ```

   The statement `open myWindow` opens the window and plays the specifed movie.

4. Press Enter or click the close box to enter the script.

You just wrote the handler that plays a Director movie in a window within another Director movie. When you are finished the handler should look like this:

```
on beginMyMovie
  global myWindow
  if objectP(myWindow) then
    forget myWindow
  end if
  set horzOrigin to the stageleft + 127
  set vertOrigin to the stageTop + 66
  set myWindowRect to rect(horzOrigin, vertOrigin, ¬
  horzOrigin + 256, vertOrigin + 192)
  set myWindow to window "wallMovie"
  set the rect of myWindow to myWindowRect
  set the fileName of myWindow to "INTERACT"
  set the titleVisible of myWindow to FALSE
  open myWindow
end
```

Finally, write the handler that closes the window when you no longer need it. In this exercise, the handler is named `finishMovie`. The calling statements for this handler have already been written for you. The handler is called when the playback head goes to the segment that plays digital video or the movie stops.

To write the `finishMovie` handler:

1. Open cast member 11, Movie Script, if it is closed.
2. Type the following at the bottom of the script window after the on stopMovie handler:

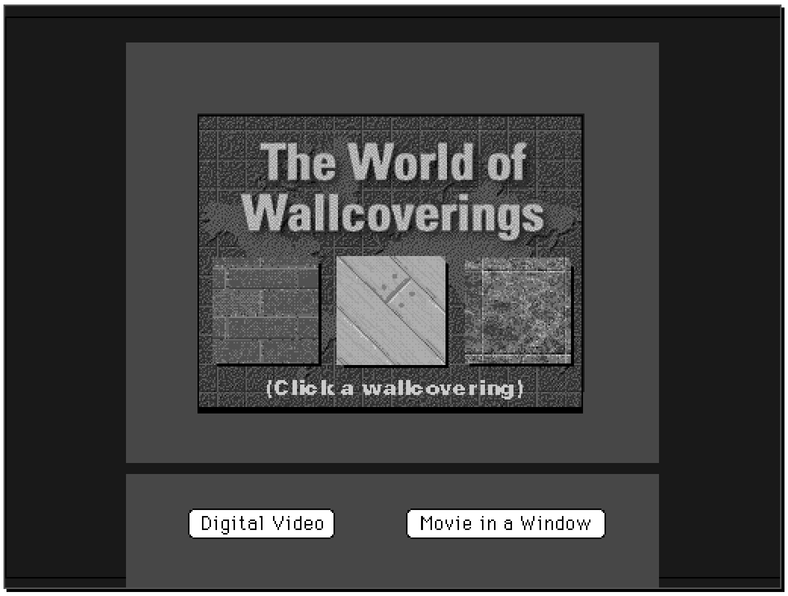
   ```
   on stopMovie
     finishMovie
   end


   on finishMovie
     global myWindow
     if objectP(myWindow) then
       forget myWindow
     end if
   end
   ```

   The `on startMovie` handler calls the `finishMovie` handler when the movie stops, The `finishMovie` handler tests whether there is window named `myWindow` and disposes of it if there is. You do not need to close the window's movie before you close the window. (The `finishMovie` handler can also be called when you click the Digital Video button. This Lingo has already been included in "MIAW.")
3. Press Enter to enter the new script.

To see the effect of the handlers you wrote.

1. Rewind and play the movie.
2. Click the Movie in a Window button.
3. The movie "INTERACT" plays in the simulated monitor screen.
   "Interact" is the filename of the interactive Director movie about wallcoverings.



4. Click the Brick, Wood, and Tile buttons.
   The movie responds to each button click.
5. Click the Digital Video button.
   The movie window closes and the digital video plays.

You have just performed the series of basic steps to implement a movie in a window. The remaining sections in this chapter provide further information about each aspect of playing movies in windows.

Controlling windows

Lingo lets you control many aspects of the window you play a movie in. Besides specifying which movie plays in the window and when the window opens and closes, you can control the behavior of the window itself and how movies in windows interact with the movie on the stage.

The following sections describe how Lingo can control windows. The features appear in the order you would typically perform them.

Setting the window type

Before the window opens, you can specify the window's type by assigning a Standard Macintosh Toolbox value for the `windowType` property. The following are the available values and the window type each specifies (the numbers 6, 7, 9, 10, 11, 13, 14, and 15 have no effect when specifying window type):

`0`	Moveable, sizeable window without zoom box
`1`	Alert box or modal dialog box
`2`	Plain box, no title bar
`3`	Plain box with shadow, no title bar
`4`	Moveable window without size box or zoom box
`5`	Moveable modal dialog box
`8`	Standard document window
`12`	Zoomable, nonresizable window
`16`	Rounded corner window

For example, the statement `set the windowType of window "Sample" to 2` sets the frame of window "Sample" to a plain box without a frame, which is the style that the number 2 specifies in the Standard Macintosh Toolbox.

It is possible to change the window type after the window is open, but this can cause a delay while the window redraws to the new type.

When you don't specify a window type, Director uses a plain box.

▶ ***Tip*** *When you use windows as part of an interface in your movie, it is a good idea to use the same window style throughout for consistent appearance.*

Opening the window

Use the `open` command to open the window any time after the window has been created. For example, the statement `open window "Noh_Tale"` opens the movie "Noh_Tale", which is the first movie of *Noh Tale to Tell*, in a window.

If the movie for the window is not in the same folder as the parent movie, you can include the element `pathName` to refer to the movie's pathname.

For example, the statement `set vMovie = the pathName ¬ & "INTERACT"` creates a variable named vMovie that contains the pathname and filename for the movie "INTERACT."

The movie is not loaded into memory until the window is first opened, which could result in a noticeable pause.

You can specify the window's appearance before or after you open the window.

Moving the window to the front or back

You can control whether a movie appears in front of or behind other windows by using the `moveToFront` and `moveToBack` commands:

- The `moveToFront` command moves the window to the front. For example, the statement `moveToFront window ¬ "Demo"` moves the window named Demo to the front of all other open windows.
- The `moveToBack` command moves the window to the back. For example, the statement `moveToBack window "Demo"` moves the window named Demo behind all other open windows.

Making the window visible

The window's `visible of window` property specifies whether the window is visible.

When the window opens, the window exists within the parent movie and is visible. You can hide a window without closing it by setting the `visible of window` property to FALSE. You can also make a window visible by setting the `visible of window` property to TRUE.

For example, the statement `set the visible of window "Sample" = TRUE` makes the window Sample visible. The statement `set the visible of window "Sample" = FALSE` makes the window Sample invisible.

▶ **Tip** *To avoid a noticeable time lag when the window opens, set the fileName of the window before it's needed and then open the window when it needs to be visible.*

For example, `set the fileName of window "Sample" = the¬ pathName & "Sample"` *loads the movie into memory and* `open window "Sample"` *makes the window visible.*

Displaying a window title

You can assign a title to the window and specify whether the title is visible. This allows you to play movies in windows that look like standard interface windows.

To assign a title to the window, set the `title of window property` to the title you want. For example, the statement `set the title of window "Sample" = "Noh Tale"` sets the window's title to "Noh Tale."

You can make the title appear by setting the `titleVisible of window` property to `TRUE`. For example, the statement `set the titleVisible of window "Sample" to TRUE` makes the title appear in the window named Sample. In our example, the title would be "Noh Tale," because you assigned that title to the window. Setting it to `FALSE` makes the window title invisible.

Interaction between windows

Movies can interact with each other using the `tell` command, which can send instructions to a separate movie. When using the `tell` command, make sure to specify which window the instructions are directed to.

For example, "Navigator" can control the playback head in the movie in the Sample window. By issuing the statement `tell window "Sample" to "go to frame 27"`, "Navigator" sends the playback head to frame 27 in *Noh Tale to Tell* when *Noh Tale to Tell* is playing in the Sample window.

When you want a movie in a window to send instructions to the primary movie, use the element `the stage` to refer to the primary movie. For example, the statement `tell the stage to go to "Help"` lets the movie in a window tell the primary movie to go to the marker named "Help."

You can prevent Director from responding to any events that occur outside the window by using the `modal of window` property. When `the modal of window` is `TRUE`, Director responds to no events outside the window, including the message window.

For example, at certain times you might want only one window to be able to respond when the user clicks the mouse or types something on the keyboard. A common case could be when you provide a help system as a movie in a window and want to make sure that nothing happens to the primary movie when the help system is playing. You can achieve this by preventing all other movies from responding when the mouse is clicked.

The statement `set the modal of window "Sample" to TRUE` does this for the Sample window. When this statement is in effect, Director responds only to events in the Sample window.

For the “Navigator,” the statement `set the modal of the stage to TRUE` lets only events in “Navigator” have effect. When this statement is in effect, Director responds to events only in “Navigator” itself but not to events in the movie playing in the Sample window.

The `modal of window` property persists after the movie stops playing. You can always close the window by pressing Command-period or clicking the window’s close box after the movie is finished.

Setting the window size and location

You can control how large the window is and where the window appears by setting the window’s screen coordinates. A window’s coordinates are given in a list in the order left, top, right, and bottom. This type of list is called a `rect`.

Setting the coordinates before the movie appears controls the initial position of the window. Setting the coordinates after the window appears moves the window.

Set the coordinates of a window by setting the rect of window property to the coordinates at which you want the window to appear. You can define the coordinates as a list or by using the rect function. For convenience, assign the coordinates to a variable and then use the variable in the statements you write.

For example, these statements set the variable `aRect` to a set of coordinates, and then apply that position to the window. The items in the list `aRect` are the rectangle's four coordinates:

```
set aRect = (0, 0, 200, 300)
set the rect of window "Sample" = aRect
```

Alternatively, you could use the rect function to define the rectangle's four coordinates, as in the following statements:

```
set aRect = rect(0, 0, 200, 300)
set the rect of window "Sample" = aRect
```

When the area defined by the coordinates assigned to `rect of window` is smaller than the movie that plays in the window, the movie is cropped.

You can pan or scale a movie by setting `the drawRect` property to coordinates smaller than the movie's original size.For example, these statements set the variable dr`awRect` to a set of coordinates, and then apply that position to the window:

```
set aRect = [0, 0, 200, 300]
set the drawRect of window "Sample" = aRect
```

Now, if the movies that play in the Sample window are larger than this rectangle, the window appears in the upper left corner and compresses the movie to fit within the rectangle.

Closing windows

Using the `close` command closes the window and makes it invisible. You can reopen the window by using the `open` command.

For example, the statement `close window "Sample"` closes the window Sample.

Using the `forget` command, you can specify that the window is no longer in use. This has the advantage that when the movie is no longer in use, Director discards the movie and removes it from memory.

For example, the statement `forget window "Sample"` has Director discard the movie when it is no longer referenced by any other window.

▶ **Tip** *When the user could possibly reopen a window by clicking on something, simply closing the window instead of forgetting it can prevent the performance loss caused by the time it would take to reload the window. Of course, closing rather than forgetting the window continues to use space in memory.*

Listing the windows

You can obtain a list of all known windows in the movie by using the `windowList` property. For example, the statement `put ¬`
`the windowList` displays a list of current window names in the message window.

Chapter 10

Parent Scripts & Child Objects

This chapter introduces parent scripts, child objects, and the ways you create them. This material is an introduction to the subject. After you understand it you should be able to go on, study, and understand the parent scripts used in *MECH*.

This chapter tells you:

- What child objects and parent scripts are
- How to create a child object from a parent script
- How to create many child objects from a parent script
- How to assign and control properties of child objects.

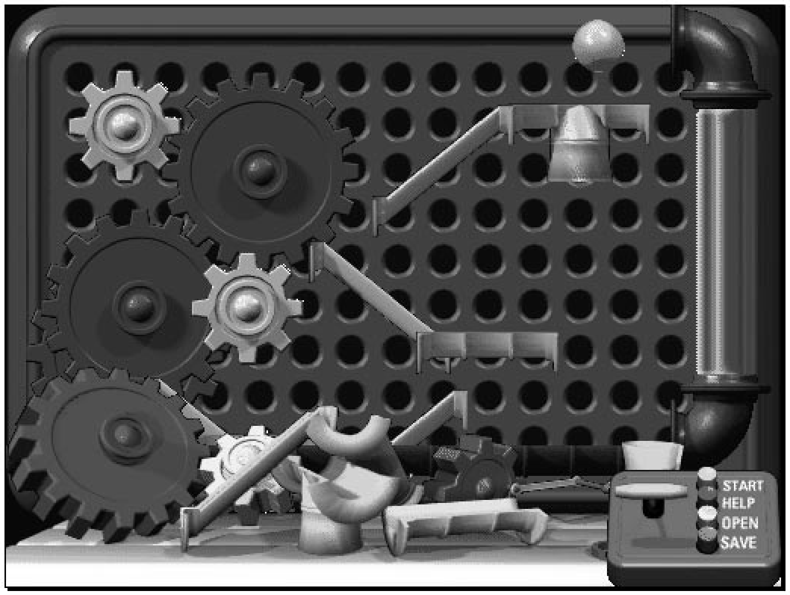
Why use child objects

Sometimes, you might want to create a set of objects that share characteristics but can still behave independently of each other. You also might want to create these objects “on demand” as the movie calls for them.

For example, you might want to create sets of interface buttons that look and behave similarly but differ in the actions they perform. You would probably also want each button in each set to be on or off independent of what happens to other buttons. You might also want to create and dispose of buttons as the movie plays.

Other times, you might want to create sprites as the movie plays and display or remove them from the stage as movie conditions require. The sprites could all look the same but move in different directions at different speeds. If two sprites collide, each could respond differently to the collision: one might reverse direction, the other might shatter.

The tools in the sample movie *MECH*, as shown below, are examples of child objects. Each item is a child object created from the same parent script. Each gear is created from the same parent script, as is each ramp and each chute. Notice that when several of the same type of item are on the pegboard, they can perform differently from each other.



As an introduction to creating child objects, this chapter shows you how to write a simple parent script to create one child object: a simple ball that you can move to the left or right by clicking buttons. In the section, “Creating multiple child objects” later in this chapter, you’ll learn how to write Lingo that creates two sets of related child objects.

Notes for experienced programmers

Parent scripts, child objects, and ancestor scripts are a simpler alternative to factories, which were used in earlier versions of Director. They are similar to classes, class instances, and inheritance used in object-oriented programming languages such as C++.

Terms that refer to ancestor scripts, parent scripts, and child objects in this guide and other Director documentation correspond to the following terms used in object-oriented programming:

Lingo term	Equivalent term
Property variable	Instance variable
Parent script	Class
Child object	Class instance
Handler	Method
Ancestor script	Super class

Using child objects, you use handlers instead of methods to define behaviors. Methods are still used by XObjects and factories.

Looking at a simple child object

Before you start writing Lingo that creates one child object from a simple parent script, look at a finished example of what you are going to make.

To see a finished example of a movie that creates a child object from a parent script:

1. Open and play the movie "SimpDone" in the Tutorials: Learning Lingo: Simulation folder.
2. Click the Birth button.
   A ball containing an image of a script appears on the stage.
3. Click the Left and Right buttons.
   The ball moves to the left or right, depending on which button you click.
4. Stop the movie and open the cast window.
5. Look at the content of cast member 8, Ball Parent Script.
   Notice that the script contains a line that starts with the `property` keyword and the handlers named `on birth` and `on moveBall`.
6. Look at the script assigned to cast member 4, the Birth button.
   The script calls the `createBall` handler.
7. Look at the createBall handler in cast member 9.
   The second statement in the handler declares ball1 a global variable. The fourth statement issues a `birth` statement. You will learn more about the `birth` statement in the next section, "Writing a parent script."
8. Close the movie when you're finished. Click Don't Save to avoid saving any unintentional changes you might have made to the movie.

The movie you just played uses a simple parent script to create the ball, which is a child object. The image of the parent script in the ball illustrates that the ball child object is actually an occurrence or instance of the set of handlers in the parent script. Clicking the Left and Right buttons activates the handlers that instruct the ball to move to the left or right.

Now see the next section "Writing a parent script," for an explanation of what a parent script contains and how to write the simple parent script used in this example.

Writing a parent script

Creating child objects requires issuing a `birth` statement—which contains the name of the parent script and any arguments that specify the child object's characteristics—to a parent script. The `birth` statement can be issued from anywhere in the movie.

`Birth` statements can continue to produce many child objects from the same parent script. When it creates a child object, Director actually creates an object that includes an identification number for the child object, a reference to the parent script, and the values assigned to any property variables that the object has. The child object is stored in RAM.

The number of objects that you can create and maintain is limited only by the amount of RAM available in the computer. The number of child objects that can be displayed on the stage is limited to the number of available sprite channels. Director maintains each child object as an item in a list.

After a child object is created, the handlers within the child object can be executed just like any other handler. Each child object can maintain its own values for its property variables and respond to messages independently of related child objects.

This section describes what to include in a parent script and shows you how to create a simple parent script of your own for the movie "Simple." Later, you'll use the parent script to create a child object: the image of the ball in the movie "SimpDone."

What's in a parent script

The parent script contains three types of Lingo:

- A `birth` handler that creates a child object each time it is called and assigns the child object its initial values
- Optional handlers that control the child object's behavior after the child object is created
- An optional statement that declares which variables are `property` variables—variables for which each child object can maintain individual values regardless of the values for other child objects.

  A special type of `property` variable that you can include is the `ancestor` property. The `ancestor` property lets a child object use handlers in an ancestor script, which is a parent script other than the one used by the child object. Ancestor scripts are discussed in the section "Creating multiple child objects," later in this chapter.

In the movie "SimpDone" that you just played, the first line declares which variables are property variables. Variables listed after `property` become property variables for the child object. In this case, the `birth` handler sets the initial values for the child object that is created. (It is generally good practice to set initial values for a child object when the child object is created.) The `moveBall` handler controls how much the ball moves to the left or right when the user clicks the Left or Right buttons.

Before you create child objects, first analyze how you want the child objects to behave. Then write a parent script that declares any appropriate property variables, includes the `birth` handler that sets up the child objects' initial values and parameters, and contains additional handlers that you decided to use.

Declaring property variables

Each child object created from the same parent script contains the same set of variables. Because individual child objects can receive different messages and behave differently after they are created, you often want each child object to maintain its own values for some of these variables, independently of what happens in a different child object. You can do this by making the variable a property variable.

You declare which variables are property variables at the beginning of the parent script by using the `property` keyword. In the movie “SimpDone,” the parent script declared the variable `horizPos` to be a property variable.

To declare this a property variable in the parent script you are writing:

1. Open the movie “Simple” in the Tutorials: Learning Lingo: Simulation folder.
2. Double-click cast member 8, Ball Parent Script, in the cast window to open it.
   Cast member 8 has already been named Ball Parent Script. The first time you open this script cast member, the script should be empty.
3. Type the following at the first line of the script:

   ```
   property horizPos
   ```
4. Click the close box in the script window to close the window and enter the script.
5. Choose Save from the File menu to save your work when you're finished.

Each property variable and its value persists as long as the object itself persists.

Note *You can access and refer to property variables from outside the child object by using the Lingo element the followed by the name of the variable. For example, the movie can issue the statement “set the horizPos of myBall to 200,” which sets the ball’s horizontal location to 200. This was not possible in earlier versions of Director.*

A property variable belongs only to the child object it is associated with. The initial value of a property variable should be set in the `birth` handler.

Creating a birth handler

Each parent script requires a birth handler—a handler that creates the new child object and sets its initial values when the handler is called. The `birth` handler always starts with the phrase `on birth`, followed by the optional `me` variable, and any arguments being passed to the new child object.

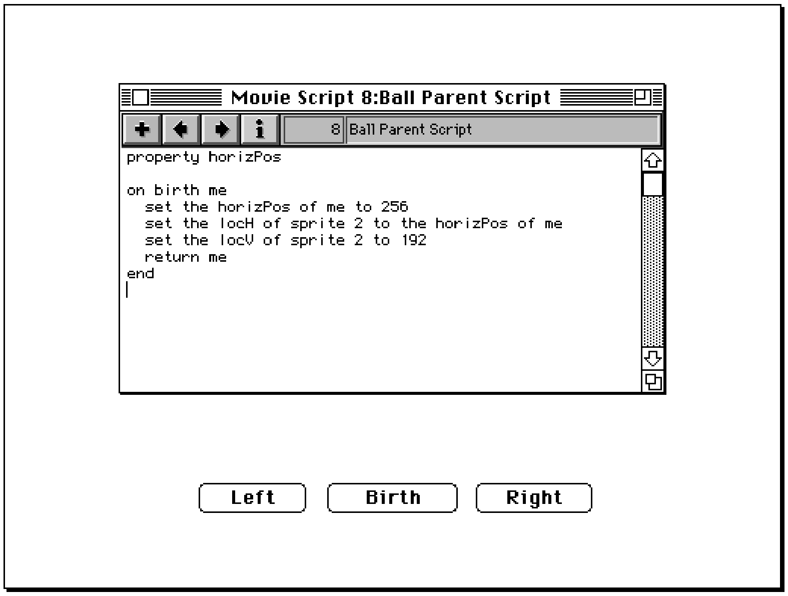
In the `birth` handler for the movie “Simple,” you use the `on birth` syntax to create a new child object and include statements that set the child object’s initial location.

To write the `birth` handler for the parent script:

1. Open cast member 8, Ball Parent Script, in the cast window of the movie “Simple” if it is closed.
   The Ball Parent Script cast member already contains the property variable statement that you wrote earlier.

2. Type the following after the first line of the script:

```
on birth me
  set the horizPos of me to 256
  set the locH of sprite 2 to the horizPos of me
  set the locV of sprite 2 to 192
  return me
end
```



```
property horizPos

on birth me
  set the horizPos of me to 256
  set the locH of sprite 2 to the horizPos of me
  set the locV of sprite 2 to 192
  return me
end
```

When the ball is created, this handler:

◇ Creates a new object when the handler is called

◇ Assigns a value to the variable `mySpeed`, which controls how much the ball moves each time you click a Left or Right button

◇ Sets the coordinates of sprite 2, the ball

◇ Makes sprite 3, the Birth button, invisible

◇ Tells Director whether the child object has been created. If the child object was created, it is added to a list of child objects that is held in memory.

3. Click the close box in the script window to close the window and enter the script.
4. Press Command-S to save your work when you're finished.

The ball is assigned to sprite 2 because the ball cast member is assigned to sprite 2 in the score. This is a very simple way to assign the new child object to a sprite channel. More complex parent scripts that you write later on use more flexible ways to assign sprite numbers.

About the me variable

The `me` variable is a common term that can be used to identify a child object When `me` is used as it is in the `birth` handler in Ball Parent Script, each child object assigns its identification number and pointer to the parent script to its version of the `me` variable. Because the `me` variable is a common term present in each handler of the child object, the `me` variable identifies each handler as part of the same child object. In a sense, it is like a family name that identifies several people as members of the same family.

You can write Lingo to assign the current instance of the child object to `me`. As a result, the `me` variable can be used to refer to whichever handler is being executed at the time, just as any person in a group of people could use the word "me" to refer to himself or herself.

The `me` variable is useful for calling a handler when you have more than one child object using the same parent script. By using `me`, you don't have to specify the individual child object each time a child object calls a handler.

The term `me` itself is used by convention. You could use any term as a common term among handlers, as long as you use it consistently. It is good practice to use `me`, because it is a simple term that is quickly identifiable by other Lingo users.

Handlers that you write here and that are used in other movies such as *MECH* are good examples of how to use the `me` variable.

Using additional handlers

You determine the child object’s behavior by including the handlers that give the behavior you want in the parent script. In the movie “SimpDone,” the `moveBall` handler moved the ball each time the handler was called when you clicked the Left or Right button. You can add this feature to the movie you’re writing by including a similar handler in the parent script.

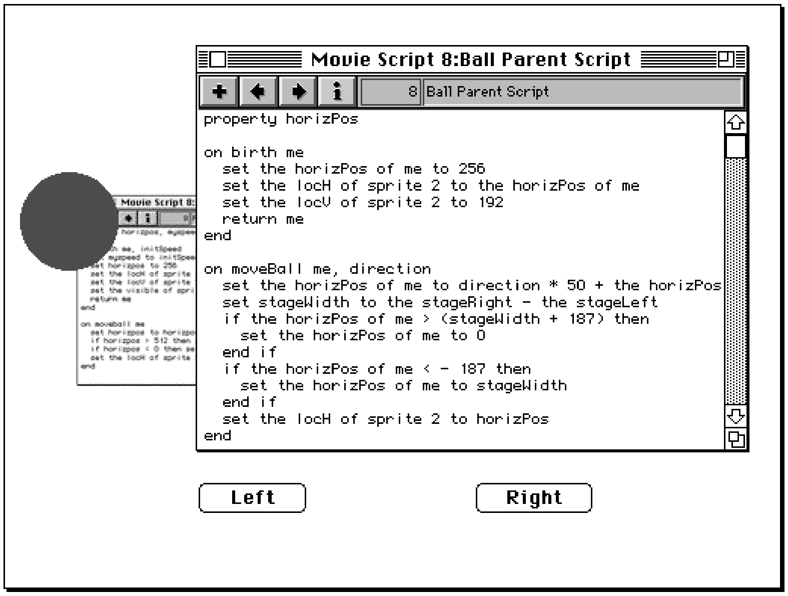
To add the `moveBall` handler to the parent script you’re writing:

1. Open the movie “Simple” if it is closed.
2. Double-click cast member 8, Ball Parent Script, in the cast window to open it.
   The Ball Parent Script cast member already contains the property variable statement and birth handler that you wrote earlier.
3. Type the following after the last line of the script:

```
on moveBall me, direction
  set the horizPos of me to direction * 50 + ¬
  the horizPos of me
  set stageWidth to the stageRight - the stageLeft
  if the horizPos of me > (stageWidth + 187) then
    set the horizPos of me to 0
  end if
  if the horizPos of me < - 187 then
    set the horizPos of me to stageWidth
  end if
  set the locH of sprite 2 to horizPos
end
```

When this handler is called, the handler moves the ball to a new location by setting the ball's `locH` to the value of `horizPos`:

- The first statement calculates the value for `horizPos` from the current value of the variables `horizPos` and `direction`. The value for `direction` comes as an argument from the handler attached to the Left or Right button.
- The second, third, and fourth statements check whether the value for `horizPos` puts the ball entirely beyond the left or right edge of the screen and resets `horizPos` so that the ball enters from the opposite edge of the stage if it is.
- The last statement moves the ball by setting its `locH` to the value of `horizPos`.



```
property horizPos

on birth me
  set the horizPos of me to 256
  set the locH of sprite 2 to the horizPos of me
  set the locV of sprite 2 to 192
  return me
end

on moveBall me, direction
  set the horizPos of me to direction * 50 + the horizPos
  set stageWidth to the stageRight - the stageLeft
  if the horizPos of me > (stageWidth + 187) then
    set the horizPos of me to 0
  end if
  if the horizPos of me < - 187 then
    set the horizPos of me to stageWidth
  end if
  set the locH of sprite 2 to horizPos
end
```

4. Click the close box in the script window to close the window and enter the script.

5. Press Command-S to save your work when you're finished.

This handler is a simple example of the types of behavior you can assign to a child object by including various handlers.

Creating child objects

You create a child object from a parent script by issuing the `birth` statement, which assigns a name to the child object and specifies the object's parent script and initial parameters. The parent script and initial parameters are defined in the `birth` function.

The `birth` statement can be issued from any script. The `birth` function has the following syntax:

```
birth(script "scriptName", argument1, argument2, ¬
 argument3...)
```

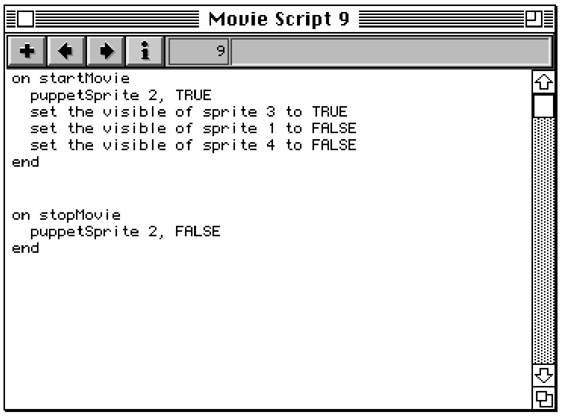
When you use the `birth` function, replace *scriptName* with the name of the parent script. Replace *argument1*, *argument2*, *argument3*... with any arguments you are passing to the child object's `birth` handler.

Note *Using the birth function differs from using mNew to create a new object using factories, although the results are similar. For information about using methods with factories, see Appendix C, "Factories."*

For the movie you're building, write a handler `createBall` that contains the `birth` statement for the ball. The Birth button already contains the calling statement for `createBall` in an `on mouseUp` handler.

To write the handler that uses the `birth` statement to create a child object:

1. Double-click cast member 9, Movie Script, in the cast window of the movie “Simple” if it is closed.
   The movie script window appears.



```
on startMovie
  puppetSprite 2, TRUE
  set the visible of sprite 3 to TRUE
  set the visible of sprite 1 to FALSE
  set the visible of sprite 4 to FALSE
end


on stopMovie
  puppetSprite 2, FALSE
end
```

2. Beneath the on startMovie handler, type the following:

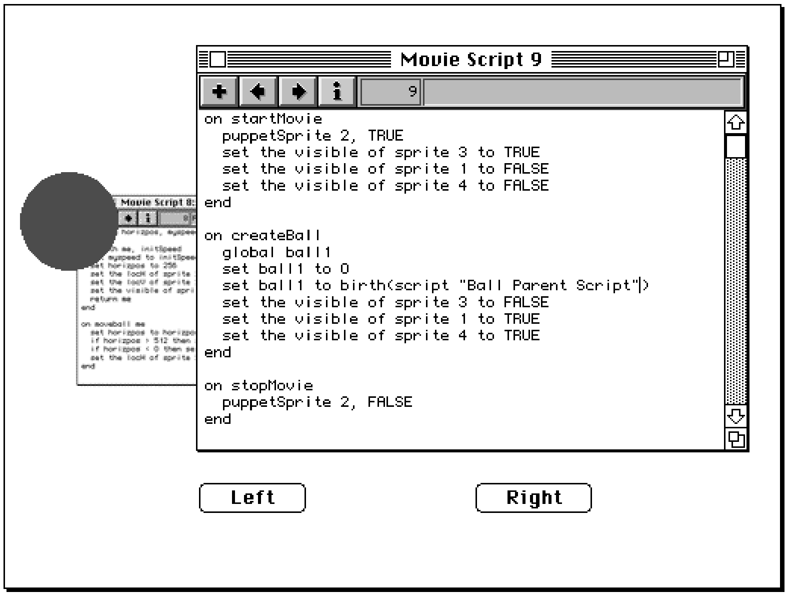
```
on createball
  global ball1
  set ball1 to birth(script "Ball Parent ¬
  Script")
  set the visible of sprite 3 to FALSE
  set the visible of sprite 1 to TRUE
  set the visible of sprite 4 to TRUE
  updateStage
end
```

   When this handler is called, it creates a child object named `ball1`. The child object is one occurrence of the parent script “Ball Parent Script” that you wrote earlier:

   ◇ The first statement sets up a global variable named `ball1` that the movie uses to refer to the ball after it is created.

- ◇ The second statement uses the `birth` statement to create the child object from the parent script and uses the `set` command to assign the name `ball1` to the child object.
- ◇ The third statement makes sprite 3, the Birth button, invisible.
- ◇ The fourth and fifth statements make sprites 1 and 4, the Left and Right buttons, visible.
- ◇ The last statement updates the stage without waiting for the playback head to enter another frame.

3. Click the close box in the script window to close the window and enter the script.
4. Rewind and play the movie.
5. Click the Create button.
   A ball, the child object, appears on the stage. (The Left and Right buttons appear but do not work yet. You will add the Lingo that makes these buttons work in the next section, "Controlling a child object.")



```
on startMovie
  puppetSprite 2, TRUE
  set the visible of sprite 3 to TRUE
  set the visible of sprite 1 to FALSE
  set the visible of sprite 4 to FALSE
end

on createBall
  global ball1
  set ball1 to 0
  set ball1 to birth(script "Ball Parent Script")
  set the visible of sprite 3 to FALSE
  set the visible of sprite 1 to TRUE
  set the visible of sprite 4 to TRUE
end

on stopMovie
  puppetSprite 2, FALSE
end
```

6. Press Command-S to save your work when you're finished.

You just wrote a handler that creates a child object from a parent script, changed the appearance of buttons, and then updated the stage.

Unless the child object has been assigned to `the actorList`, you can remove a child object from the movie by setting the child object to zero. For example, the statement `set ball1 = 0` would remove the child object `ball1` from the movie. For information about `the actorList` see the *Lingo Dictionary*.

Controlling a child object

Lingo can control a child object by sending messages to handlers the same way it sends messages to handlers anywhere else in the movie.

In the movie “SimpDone,” clicking the Left or Right button moves the ball to the left or right. This occurs because scripts attached to the buttons call the `moveBall` handler that is in the parent script. In turn, the `moveBall` handler moves the child object each time the user presses the mouse button.

For the movie you are building, the `moveBall` handler is already written in the parent script. You will write the calling statements for the `moveBall` handler and assign them to the Left and Right buttons. When one of these buttons is pressed, it calls the `moveBall` handler in the parent script and passes it the appropriate value for the variable `direction`.

Whether the value of the argument `direction` is 1 or –1 determines whether motion is to the left or right.

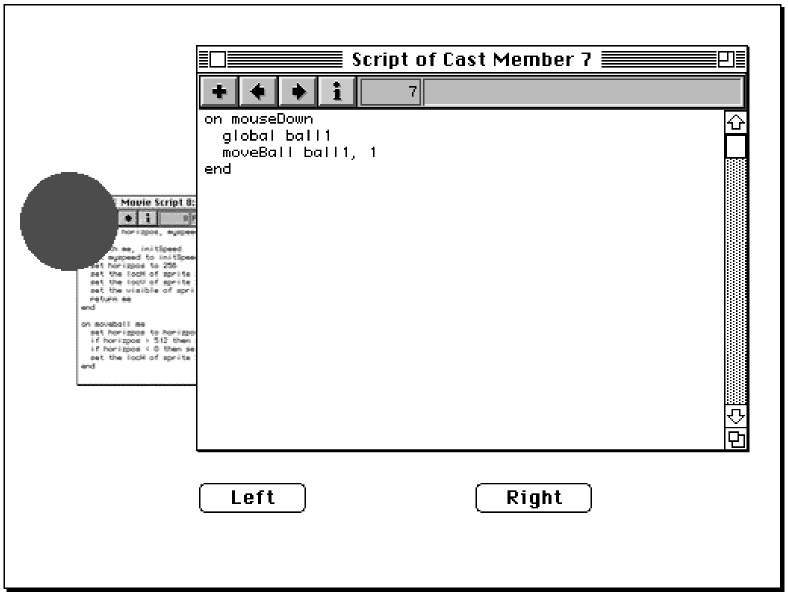
To write the handler for the Left button:

1. Open the movie “Simple” in the Tutorials: Learning Lingo: Simulation folder.
2. Double-click cast member 10, Left, in the cast window to open its button window.
3. Click the script button in the button window.
   The script of cast member window appears.
4. Delete the line `on mouseUp` and then type the following before the line `end`:
   ```
   on mouseDown
     global ball1
     moveBall ball1, -1
   ```
5. Click the close box in the script window to close the window and enter the script.
6. Rewind and play the movie.
7. Click and hold the Left button.
   The ball moves to the left when you click the Left button.

To write the handler for the Right button:

1. Double-click cast member 7, Right, in the cast window to open its button window.
2. Click the script button in the button window.
   The script of cast member window appears.
3. Delete the line `on mouseUp` and then type the following before the line `end`:

```
on mouseDown
  global ball1
  moveBall ball1, 1
```



```
on mouseDown
  global ball1
  moveBall ball1, 1
end
```

4. Click the close box in the script window to close the window and enter the script.
5. Rewind and play the movie.
6. Click and hold the Right button.
   The ball moves to the right each time you click the Right button.
7. Press Command-S to save your work when you're finished.

You just wrote two handlers that move the ball to the left or right. When the Left or Right button is clicked, the sprite's `on mouse`Down handler calls the `moveBall` handler in the parent script. As a result the ball moves according to the instructions in the `moveBall` handler. The ball moves left or right depending on whether the variable `direction` is positive or negative.

Looking at multiple objects

The movie that you built earlier in this chapter was a very simple example meant to show you how create a child object from a parent script. In normal use, you would not use a parent script to create just one child object.

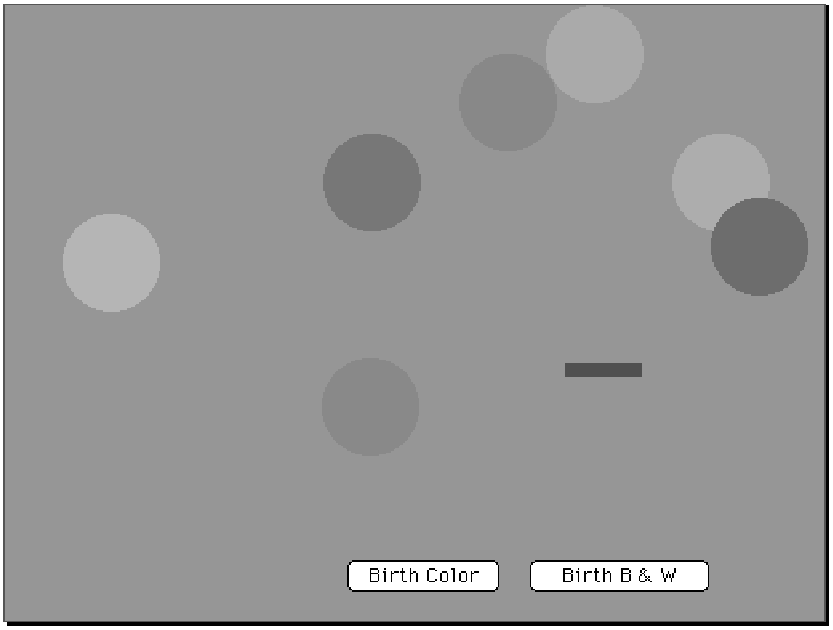
The advantage of parent scripts and child objects comes from their ability to:

- Create many child objects on command as the movie plays. You create a child object each time you issue a `birth` statement.
- Maintain individual behaviors for each child object.

This and later sections show you how to write sample Lingo that creates similar sets of objects from different parent scripts.

To see a finished version of the movie you'll build:

1. Open and play the movie "PareDone" in the Tutorials: Learning Lingo: Simulation folder.
2. Click the Birth Color and Birth B & W buttons several times.
   Each time you click a button, a new ball appears on the stage. Clicking the Birth Color button makes a color ball; clicking the Birth B &W button creates a grayscale ball. The balls fall unless they hit the blue rectangle. If the balls hit the blue rectangle, they reverse direction.
3. Click different places on the stage several times so that the rectangle is under different balls.
   The rectangle moves to where you click. Each time a falling ball strikes the rectangle, the ball changes direction. This is what a typical screen looks like after clicking the buttons several times:



4. Close the movie when you're finished.
   Click Don't Save to avoid saving any unintentional changes you might have made to the movie.

The movie you just played used a parent script to create bouncing balls, which are child objects. Unlike the movie “Simple” that you played earlier in this chapter, “PareDone” creates more than one child object and allows the child objects to have individual behaviors because they maintain their own property variables. The similarities in the balls come from an ancestor script. (Ancestor scripts are described in the next section, “Creating multiple child objects.”) The differences—color, in this case—come from different parent scripts:

- All balls follow the same rules for motion because this characteristic is inherited from the ancestor.
- Each ball can have a different velocity, direction, and sprite number because these are declared to be property variables in the ancestor script.
- The balls are either color or grayscale, depending on which parent script they have.

Creating multiple child objects

You create a child object each time you issue the `birth` statement. But when you have more than one child object, it is very important to identify which child objects have been created and assign them sprite numbers when you want them to appear on the stage.

For simple situations, you can often track child objects and assign them sprite numbers by maintaining a list of child objects and developing a straightforward routine that assigns sprite numbers. The movie "PareDone" maintains a list that can track up to ten child objects. This is the approach that you use here. For examples of how you can track child objects in more complex movies, see the movie *MECH*.

Sets of child objects can also differ from one another, the way a group of automobiles can share characteristics but differ from each other in some ways. For example, all automobiles in a group could have four wheels and a windshield. However, some automobiles could have four doors instead of two doors. Each automobile could be painted a different color. Of course, when it is being driven, each automobile can be driven at different speeds, regardless of how fast the other automobiles are going.

In the automobile example, four wheels and a windshield are common characteristics. These are similar to the types of characteristics set by ancestor scripts. The number of doors is a characteristic that could be assigned by different parent scripts:

- ◆ Using the ancestor script with the parent script that specifies four doors creates an automobile with four doors, four wheels, and a windshield.
- ◆ Using the ancestor script with the parent script that specifies two doors creates an automobile with two doors, four wheels, and a windshield.

Color and speed resemble property variables, which are individual values for each child object.

Lingo lets you create similar situations for groups of child objects by using ancestor scripts, which are an additional source of handlers available to a child object. The child objects of different parent scripts can share characteristics defined by the ancestor script. You assign a child object a different behavior than the one in the ancestor script by giving the child object's parent script a handler that overrides the equivalent handler in the ancestor script.

For example, suppose a group of cousins all inherit baldness from a common ancestor on one side of the family. All cousins would be bald if no opposing trait came from the other parent. However, one group of brothers and sisters could still inherit full heads of hair if that trait was a characteristic of the other parent. In this case, baldness is similar to a characteristic received from an ancestor script. Full heads of hair in one group of brothers and sisters is similar to a characteristic received from a parent script: the parent script "intercepts" or overrides baldness received from the ancestor script.

In this section, you re-create Lingo that creates multiple child objects from the same ancestor script in the movie "PareDone."

The child objects are balls that can move according to the same rules but each responds to collisions independently of the other. The balls receive the same set of rules that govern their motion in handlers from an ancestor script. They also have the same property variables. Rules that determine their color are in handlers that come from separate parent scripts. The balls can have different colors and change their motion independently of the other balls.

By adding more handlers to a parent script or ancestor script, you could control additional behaviors of the balls. After you master the technique of producing balls from the simple parent script, analyze the ancestor and parent scripts in the movie *MECH* for ideas about additional uses for child objects.

However, before you start, look at the existing handlers in "Parents," the movie you are going to add Lingo to, to see how the Lingo has been structured for this movie.

Understanding the parent scripts

The parent scripts are already written for you.

To study the parent scripts:

1. Open the movie "Parents" in the Tutorials: Learning Lingo: Simulation folder.
2. Double-click cast member 4, BW Parent, in the cast window to open it.
   This is the parent script for the grayscale balls.
3. Open script cast member 3, Color Parent.
   This is the parent script for the colored balls.
4. Study the Lingo and the comments in the parent scripts.
   Each parent script contains:
   - The `property ancestor` statement, which lets the child object use handlers in the ancestor script in addition to handlers in the parent script.
   - A `birth` handler that contains its own `birth` statement that calls the ancestor script.
   - Statements that assign a color to the ball.
5. Close the movie when you're finished. Click Don't Save to avoid saving any unintentional changes you might have made to the movie.

Setting a behavior in the parent script overrides the behavior's setting in the `ancestor` script. This hierarchy is similar to the way a sprite script takes precedence over the script of a cast member.

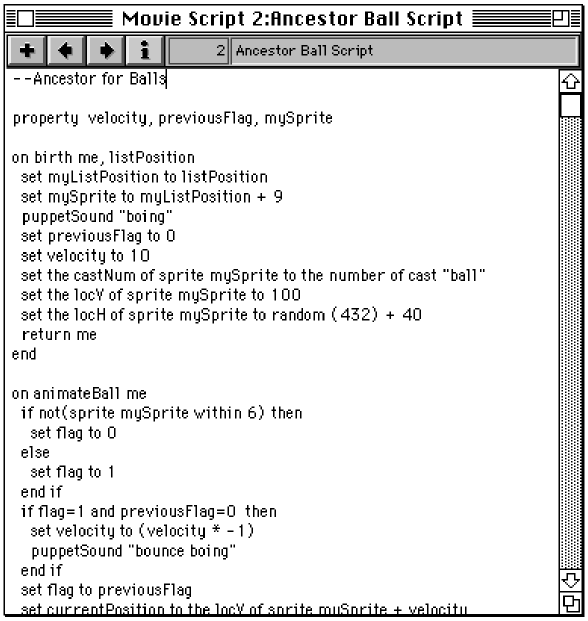
For example, in the Color Parent and BW Parent scripts, foreground color is set in the parent scripts. These child objects would receive the foreground color assigned by the parent script regardless of what the ancestor script specified. Using parent scripts this way lets you make groups of child objects that share some characteristics but differ in others, similar to shared traits among cousins as described in the section "Why use child objects," earlier in this chapter.

Understanding the ancestor script

The ancestor script has already been written for you.

To study the ancestor script:

1. Open the movie "Parents" in the Tutorials: Learning Lingo: Simulation folder.
2. Double-click cast member 2, Ancestor Ball Script, in the cast window to open it.
   This is the ancestor script for the balls.



```
--Ancestor for Balls

property  velocity, previousFlag, mySprite

on birth me, listPosition
  set myListPosition to listPosition
  set mySprite to myListPosition + 9
  puppetSound "boing"
  set previousFlag to 0
  set velocity to 10
  set the castNum of sprite mySprite to the number of cast "ball"
  set the locV of sprite mySprite to 100
  set the locH of sprite mySprite to random (432) + 40
  return me
end

on animateBall me
  if not(sprite mySprite within 6) then
    set flag to 0
  else
    set flag to 1
  end if
  if flag=1 and previousFlag=0  then
    set velocity to (velocity * -1)
    puppetSound "bounce boing"
  end if
  set flag to previousFlag
  set currentPosition to the locV of sprite mySprite + velocity
```

3. Study the Lingo and the comments in the ancestor script.
   The ancestor script contains two handlers:

   ◇ The `on birth` handler creates the child object and assigns it a list position, sprite number based on the list position, a cast member for the sprite, and an initial location.

   ◇ The `on animateBall` handler controls the ball's motion. The `animateBall` handler is called when the ball strikes the blue paddle.

4. Close the movie when you're finished. Click Don't Save to avoid saving any unintentional changes you might have made to the movie.

Understanding the movie script

Much of the needed movie script is already written for you.

To study the movie script:

1. Open the movie "Parents" in the Tutorials: Learning Lingo: Simulation folder.

2. Double-click cast member 1, Movie Script, in the cast window to open it.

3. Study the Lingo and the comments in the movie script. The movie script for the movie "Parent" contains:

   ◇ The `on startMovie` handler, which initializes the movie's global variables and turns off any puppet sprites. Later, you'll add a line that initializes the list that contains ball child objects.

   ◇ The `whichButton` handler, which determines whether the user clicked the Birth Color button, Birth B&W button, or the stage, thus moving the blue paddle. Clicking one of the Birth buttons executes the createBall handler and sends createBall the argument 0 or 1, which identifies the button that was clicked.

   ◇ The `placePaddle` handler, which relocates the blue paddle to the place that the user clicks on the stage.

   ◇ The `createBall` handler, which uses the argument `whichType` to indicate which button was clicked and issue a birth statement to the appropriate parent script.

   ◇ The `animate` handler, which determines whether balls are in the list and then calls the `animateBall` to motion for any balls that are in the list.

4. Close the movie when you're finished. Click Don't Save to avoid saving any unintentional changes you might have made to the movie.

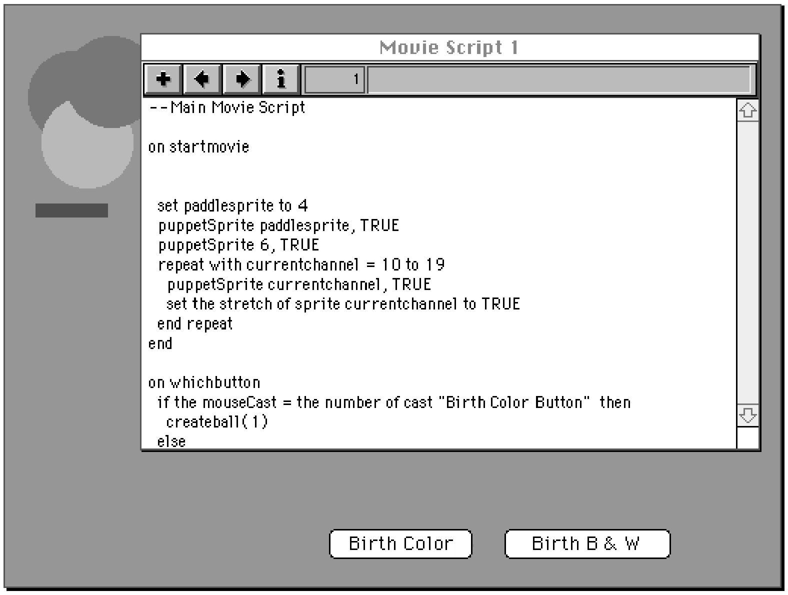
Setting up a list for child objects

A list is a useful way to track the child objects that are currently in a movie. The movie "Parents" tracks the balls in a list called `ballList`.

To set up the list in the movie you're building:

1. Double-click cast member 1, Movie Script, in the cast window for the movie "Parents" to open the script window.
   The script window appears.
2. Type the following lines immediately following the line `on startMovie`:

   ```
   global ballList, paddleSprite
   set ballList to []
   ```



```
--Main Movie Script

on startmovie


 set paddlesprite to 4
 puppetSprite paddlesprite, TRUE
 puppetSprite 6, TRUE
 repeat with currentchannel = 10 to 19
  puppetSprite currentchannel, TRUE
  set the stretch of sprite currentchannel to TRUE
 end repeat
end

on whichbutton
 if the mouseCast = the number of cast "Birth Color Button"  then
  createball(1)
 else
```

3. Click the close box in the script window to close the window and enter the script.
4. Press Command-S to save your work when you're finished.

You just wrote statements that declare `ballList` and `paddleSprite` to be global variables and then define `ballList` as a list that has no elements yet.

Adding the handler that creates the balls

Clicking Birth Color or Birth B&W in the movie "PareDone" created a new ball by first calling the handler `whichButton` when the user clicks the stage. If one of the buttons was clicked, the `createBall` handler in the movie script was called.

To add the `createBall` handler to the movie you are building:

1. Double-click cast member 1, Movie Script, in the cast window for the movie "Parents" to open the script window.
   The script window appears.
2. Type the following handler in the movie script:

```
on createBall whichType
  global ball1, ballList
  if count(ballList) >=10 then
    beep
  else
    if whichType = 1 then
      add(ballList, birth(script ¬
      "Color Parent", count(ballList) + 1))
    else
      add(ballList, birth(script "BW Parent",¬
      count(ballList) + 1))
    end if
  end if
end
```

This handler first declares `ball1` and `whichType` to be global variables. The handler then checks whether there are ten or more elements in the list named `ballList`:

- ◇ When there are more than ten child objects, the computer beeps and adds no more elements to the list. Limiting the size of the list restricts the maximum number of child objects to ten. For larger numbers of objects, you could use approaches similar to those used in *MECH*.
- ◇ When there are fewer than ten child objects, the handler uses the value of `whichType` to determine whether the Birth Color or Birth B&W button was clicked. Then the handler adds a new child object to the appropriate list by using the `add` command. The `birth` function defines the child object being added. The value `count(ballList) + 1` is an argument passed to the `birth` handler.

3. Click the close box in the script window to close the window and enter the script.
4. Press Command-S to save your work when you're finished.

Testing what you have written

You have just written several examples of Lingo that create more than one child object from the same parent script and use a list to track which child objects exist in the movie.

To test what you have done:

1. Rewind and play the movie "Parents."
2. Click the Birth Color and Birth BW buttons several times.
   Each time you click the button, a new ball appears on the stage. Clicking Birth Color produces a colored ball; clicking Birth BW produces a grayscale ball. The balls fall unless they hit the blue rectangle.
3. Click different stage locations to relocate the rectangle so that it is under different balls.
   Each time a falling ball strikes the rectangle, the ball changes direction.
4. Close the movie when you're finished.

The Lingo you added creates the balls from the parent script and gives the movie a way to track which objects are currently in the movie.

Studying MECH

This chapter exposed you to ways that you can create and track multiple child objects. For examples of advanced uses for parent scripts and child objects, see the parent scripts in *MECH*.

Ancestor and parent scripts in *MECH* produce child objects that have more complex behavior and occur in greater numbers than the parent scripts you studied in this chapter. For an explanation of what the handlers in *MECH* do, study the comments that are part of the scripts.

MECH sometimes uses `the actorList` property to support animation for child objects by sending each object attached to the list a `stepFrame` message when the playback head advances. This is a simpler alternative to the `perFrameHook` property that was used in previous versions of Lingo. For more information about `the actorList`, see the `actorList` entry in the *Lingo Dictionary*.

Appendix A

Using XCMDs and XFCNs

This appendix tells you:

- What XCMDs and XFDCNs are and how they differ from XObjects
- How to use XCMDs and XFCNs
- How to use callbacks for XCMDs.

Using XCMDs and XFCNs in Director

Lingo lets you use HyperCard's XCMDs and XFCNs in your movies. Using `XCMDGlue`—part of Director's `*Standard.xlib` library of XObjects— you can access XCMDs and XFCNs from Lingo scripts. This lets you can extend Director's capabilities by using the many XCMDs and XFCNs available from HyperCard.

Most XCMDs and XFCNs work automatically with `XCMDGlue`, but some may not.When the XCMD's primary purpose is to perform a HyperCard-specific action—such as handling cards, HyperTalk scripts, or other parts of the HyperCard interface—the XCMD or XFCN might generate an error message when used in Director.

XCMDs and XFCNs are closely related. For convenience, this appendix refers to them collectively as XCMDs.

Note *XCMDs provide an interface to external code modules but are not capable of ensuring that the external code modules themselves perform as intended. You must make sure that the external code modules perform correctly to have them produce the desired results in Director.*

Differences between XObjects and XCMDs

`XCMDGlue` works differently from XObjects. You don't create instances of `XCMDGlue` to work with specific XCMDs. Instead, `XCMDGlue` acts as an interpreter between Lingo and the XCMD.

A major difference between XCMDs and XObjects is that an XObject can have multiple instances:

- ◆ One XObject can be used to create a number of independent objects, each capable of performing different operations.
- ◆ An XCMD cannot create new instances, so it can perform only one function at a time. For more information about XObjects and instances, see Appendix B, "Using XObjects."

For these cases, you can use Lingo to create a special mechanism which may solve the problem. For further information, see the section "XCMDs and callbacks," later in this appendix.

Learning to use XCMDs

Like using XObjects, using an XCMD involves three basic steps:

1. Opening the XCMD
2. Exchanging messages with the XCMD to perform some function
3. Closing the XCMD.

One of the best ways to learn about XCMDs is to use them in Director's message window. In this section, you'll see how to open, view the contents of an XCMD resource by exchanging a message with the XCMD, and close an XCMD.

Opening XCMD resources

XCMDs can be located in two places: in an external file or in a Director movie.

When an XCMD resource is stored in the current movie's resource fork, the XCMD is automatically opened when the movie is opened. This is similar to the way `*Standard.xlib` is automatically opened when you launch Director. You can copy XCMD resources into your Director movie using a resource editor like ResEdit.

When an XCMD resource is stored in an external file such as a resource file or stack, you can open it with the `openXlib` command. If the file is in another folder, you must specify a full pathname to the folder. The easiest way to access the file is to place it in the same folder as your Director movie or the Director application.

To open an XCMD using the `openXlib` command:

1. Launch Director.
2. Open the message window and type `openXlib` followed by the name of the XCMD resource file.
3. Press Return.
   The resource file you specified opens.

One resource file can contain multiple XCMDs. When you use the `openXlib` command, all XCMDs stored in the specified XCMD resource file are opened. The XCMD resource file can be a HyperCard stack, a resource file, or even a TeachText document containing XCMD resources. Notice that this is the same command used to open regular XObjects.

Viewing XCMD resources

After you've opened the XCMD, you can use the `showXlib` command to display all open resource files that contain XCMDs as well as XObjects.

To display a list of all open resource files that contain XCMDs and XObjects:

▶ Type `showXlib` in the message window, and then press Return.

To display the contents of a specific XCMD resource file:

▶ Type `showXlib` followed by the name of the resource file, and then press Return.

Closing XCMD resources

The `closeXlib` command lets you close all open resource files that contain XCMDs and XObjects.

To close all open resource files that contain XCMDs and XObjects:

▶ Type `closeXlib` in the message window, and then press Return.

To close a specific resource file that contains XCMDs:

▶ Type `closeXlib` followed by the name of the resource file in the message window, and then press Return.

Using an XCMD or XFCN

In many cases, once you open an XCMD, you can use the XCMD in your Lingo scripts the same way you would use it in a HyperTalk script. `XCMDGlue` does everything else by converting the XCMD for you. For example, the following handler would let you use the `MIDIplay` XCMD (from Opcode Systems) to play a MIDI file from Director:

```
on startMIDIplayback
  openXlib (the pathname & "MIDIplay")
  -- opens the MIDIplay XCMD
  -- Use Lingo's "pathname" function to find
  -- resource files
  -- in the same folder as your movie
  MIDIplay "open","MyDrive:MyFolder:myMIDIfile"
  -- opens the MIDI file to be played
  MIDIplay "start"
  -- starts playback of the MIDI file
end startMIDIplayback
```

This handler stops the playback of the MIDI file:

```
on stopMIDIplayback
  MIDIplay "stop"
  closeXlib (the pathname & "MIDIplay")
end stopMIDIplayback
```

XCMDs and callbacks

Not all XCMDs can be used with `XCMDGlue` in a completely transparent manner. Occasionally, `XCMDGlue` is unable to properly convert the XCMD. When you attempt to use an XCMD's syntax in a script, an error message is displayed.

Certain XCMDs may call on HyperCard to internally perform some tasks while the XCMD is executing. Most of these are conversion routines and are used to conveniently convert information to and from different formats. The remaining callbacks either involve the HyperTalk interpreter or access information stored in HyperCard–specific entities such as fields, or they do both. The table of HyperCard callback requests at the end of this appendix lists specific technical information regarding these callbacks.

Lingo automatically supports all callbacks that are not overly specific to HyperCard. Still, some HyperCard–specific callbacks are supported when Lingo provides a direct equivalent. The remaining callbacks that are not automatically supported (a total of nine) are so specific to HyperCard that they cannot be resolved automatically unless the application calling the XCMD is virtually identical to HyperCard. Even in such cases, it is still possible to use an XCMD by using a user–defined mechanism called a callback handler.

Using a callback handler

A callback handler uses a Lingo factory to accept and respond to messages that correspond to HyperCard callback requests. A factory is a set of scripts that can be used to create an object. In Director 4.0, the functionality of factories has largely been replaced by parent scripts. For more information on parent scripts, see Chapter 10, "Parent Scripts and Child Objects." In this specific case, however, a factory provides the best way to respond to callbacks. This section shows you the steps necessary to create a callback factory, and to call that factory from a handler.

Essentially, a callback handler provides a mechanism that some XCMDs already expect to be available. The XCMD expects that when it sends or receives a callback message, something will be there to receive it and possibly return another message. (Usually HyperCard does this.) A callback handler defined in Lingo simply intercepts and returns these messages when appropriate. Whether you choose to use this information depends on your understanding of the purpose of the callback.

Fortunately, when `XCMDGlue` does not understand a callback request, it indicates the name of the callback in the error message. Once you know which callback your XCMD needs to deal with, you can create a callback handler for it. There are three basic steps to creating a callback handler:

1. Defining a callback factory
2. Creating the callback object
3. Specifying the XCMD to be used with the callback object (with the setCallBack command that is part of XCMDGlue).

Defining the callback factory

The first step in creating a callback factory is to define it. The following example factory includes methods for all the callbacks that are not supported by `XCMDGlue`. This factory does not attempt to do anything with the callback requests other than create a record of them in the message window. As you'll see later, you can use this information to process callbacks. This factory should be placed in a movie script:

```
factory callBackFactory
method mNew
me(mPut, 1, "SendCardMessage")
me(mPut, 2, "EvalExpr")
me(mPut, 3, "StringLength")
me(mPut, 4, "StringMatch")
me(mPut, 5, "SendHCMessage")
me(mPut, 6, "ZeroBytes")
me(mPut, 7, "PasToZero")
me(mPut, 8, "ZeroToPas")
me(mPut, 9, "StrToLong")
me(mPut, 10, "StrToNum")
me(mPut, 11, "StrToBool")
me(mPut, 12, "StrToExt")
me(mPut, 13, "LongToStr")
me(mPut, 14, "NumToStr")
me(mPut, 15, "NumToHex")
me(mPut, 16, "BoolToStr")
me(mPut, 17, "ExtToStr")
me(mPut, 18, "GetGlobal")
me(mPut, 19, "SetGlobal")
me(mPut, 20, "GetFieldByName")
me(mPut, 21, "GetFieldByNum")
me(mPut, 22, "GetFieldByID")
me(mPut, 23, "SetFieldByName")
me(mPut, 24, "SetFieldByNum")
me(mPut, 25, "SetFieldByID")
me(mPut, 26, "StringEqual")
me(mPut, 27, "ReturnToPas")
me(mPut, 28, "ScanToReturn")
me(mPut, 31, "FormatScript")
me(mPut, 32, "ZeroTermHandle")
me(mPut, 33, "PrintTEHandle")
me(mPut, 34, "SendHCEvent")
me(mPut, 35, "HCWordBreakProc")
me(mPut, 36, "BeginXSound")
me(mPut, 37, "EndXSound")
me(mPut, 38, "RunHandler")
me(mPut, 39, "ScanToZero")
me(mPut, 40, "GetXResInfo")
me(mPut, 41, "GetFilePath")
me(mPut, 42, "FrontDocWindow")
me(mPut, 43, "PointToStr")
me(mPut, 44, "RectToStr")
me(mPut, 45, "StrToPoint")
me(mPut, 46, "StrToRect")
me(mPut, 47, "GetFieldTE")
me(mPut, 48, "SetFieldTE")
me(mPut, 49, "GetObjectName")
me(mPut, 50, "GetObjectScript")
me(mPut, 51, "SetObjectScript")
me(mPut, 52, "StackNameToNum")
me(mPut, 53, "Notify")
me(mPut, 54, "ShowHCAlert")
me(mPut, 100, "NewXWindow/GetNewXWindow")
me(mPut, 101, "CloseXWindow")
me(mPut, 102, "SetXWIdleTime")
me(mPut, 103, "XWHasInterruptCode")
me(mPut, 104, "RegisterXWMenu")
me(mPut, 105, "BeginXWEdit/EndXWEdit")
me(mPut, 106, "SaveXWScript")
me(mPut, 107, "GetCheckPoints")
me(mPut, 108, "SetCheckPoints")
me(mPut, 109, "XWAllowReEntrancy")
me(mPut, 110, "SendWindowMessage")
me(mPut, 111, "HideHCPalettes")
me(mPut, 112, "ShowHCPalettes")
me(mPut, 113, "XWAlwaysMoveHigh")
me(mPut, 200, "GoScript")
me(mPut, 201, "StepScript")
me(mPut, 202, "AbortScript")
me(mPut, 203, "CountHandlers")
me(mPut, 204, "GetHandlerInfo")
me(mPut, 205, "GetVarInfo")
me(mPut, 206, "SetVarValue")
me(mPut, 207, "GetStackCrawl")
me(mPut, 208, "TraceScript")

method mEvalExpr x
  put "mEvalExpr:" && x

method mSendHCMessage x
  put "mSendHCMessage:" && x

method mSendCardMessage x
  put "mSendCardMessage:" && x

method mGetFieldByName card, name
  put "mGetFieldByName:" && card && name

method mGetFieldByNum card, num
  put "mGetFieldByNum:" && card && num

method mGetFieldByID card, id
  put "mGetFieldByID:" && card && id

method mSetFieldByName card, name, value
  put "mSetFieldByName:" && card && name && value

method mSetFieldByNum card, num, value
  put "mSetFieldByNum:" && card && num && value

method mSetFieldByID card, id, value
  put "mSetFieldByID:" && card && id && value

method mUnknown which
  put me(mGet, value(which)) into callBackName
  put "mUnknown:" && which && "(" & ¬
  callbackName & ")"
```

You do not need to specify every callback handled in this factory. You are required to define methods only for the callbacks that are indicated in error dialogs generated by the XCMD. For example, the `mEvalExpr` callback may be the only callback you need to account for.

As indicated in this example, the `put` statements in each method are optional. They are there to let you know what the XCMD or XFCN is attempting to tell HyperCard. You can use this information in any way you want. Sometimes, a callback requires a value (message) to be sent back to HyperCard. If you know what that value should be, use `return` at the end of the specific callback method's script. For example, if a callback required HyperCard to return `TRUE` or `FALSE` you could use a method similar to the following:

```
method callBackMethod
  if test then return TRUE else return FALSE
end callBackMethod
```

Some XCMDs use a large amount of processor time. In this situation, using a `put` statement in your script slows down whatever the XCMD does, because the `put` statement has to be evaluated and written into the message window. You can optimize the callback factory in this case by removing the `put` statements.

When a callback error occurs, the XCMD usually stops running after you click OK in the error dialog box. However, because of the design of certain XCMDs, the XCMD sometimes continues to execute. You still need to create a callback handler for these XCMDs. Otherwise, unexpected results could occur.

Creating the callback object

After you have defined a callback factory, you can create a factory object using the following statement:

```
put callbackFactory(mNew) into callbackObject
```

Specifying the callback handler

Finally, you specify the callback handler with the following statement:

```
setCallBack XCMD/XFCNname, callbackObject
```

The `setCallBack` command is part of the `XCMDGlue` XObject.

The XCMD or XFCN should now function properly. If you later use other elements of the XCMD's syntax, you might still need to deal with other callbacks. You can accomplish this easily by adding the appropriate method to your callback factory.

XCMD and XFCN callback requests

The following are HyperCard's callback requests. The symbol in the rightmost column identifies which level of support is provided for each callback.

HyperCard's callback requests

Number	HyperCard callback	Type
1	SendCardMessage	✍
2	EvalExpr	✍
3	StringLength	✔
4	StringMatch	✔
5	SendHCMessage	✍
6	ZeroBytes	✔
7	PasToZero	✔
8	ZeroToPas	✔
9	StrToLong	✔
10	StrToNum	✔
11	StrToBool	✔
12	StrToExt	✔
13	LongToStr	✔
14	NumToStr	✔
15	NumToHex	✔
16	BoolToStr	✔
17	ExtToStr	✔
18	GetGlobal	✔
19	SetGlobal	✔

HyperCard’s callback requests

Number	HyperCard callback	Type
20	GetFieldByName	✍
21	GetFieldByNum	✍
22	GetFieldByID	✍
23	SetFieldByName	✍
24	SetFieldByNum	✍
25	SetFieldByID	✍
26	StringEqual	✔
27	ReturnToPas	✔
28	ScanToReturn	✔
31	FormatScript	✍
32	ZeroTermHandle	✍
33	PrintTEHandle	✍
34	SendHCEvent	✍
35	HCWordBreakProc	✍
36	BeginXSound	✍
37	EndXSound	✍
38	RunHandler	✍
39	ScanToZero	✔
40	GetXResInfo	✍
41	GetFilePath	✍
42	FrontDocWindow	✍
43	PointToStr	✍
44	RectToStr	✍
45	StrToPoint	✍
46	StrToRect	✍

HyperCard’s callback requests

Number	HyperCard callback	Type
47	GetFieldTE	✍
48	SetFieldTE	✍
49	GetObjectName	✍
50	GetObjectScript	✍
51	SetObjectScript	✍
52	StackNameToNum	✍
53	Notify	✍
54	ShowHCAlert	✍
100	NewXWindow/ GetNewXWindow	✍
101	CloseXWindow	✍
102	SetXWIdleTime	✍
103	XWHasInterruptCode	✍
104	RegisterXWMenu	✍
105	BeginXWEdit/EndXWEdit	✍
106	SaveXWScript	✍
107	GetCheckPoints	✍
108	SetCheckPoints	✍
109	XWAllowReEntrancy	✍
110	SendWindowMessage	✍
111	HideHCPalettes	✍
112	ShowHCPalettes	✍
113	XWAlwaysMoveHigh	✍
200	GoScript	✍
201	StepScript	✍

HyperCard’s callback requests

Number	HyperCard callback	Type
202	AbortScript	✍
203	CountHandlers	✍
204	GetHandlerInfo	✍
205	GetVarInfo	✍
206	SetVarValue	✍
207	GetStackCrawl	✍
208	TraceScript	✍

✔: Automatically supported by Lingo.

✍: Requires a callback handler. Some messages and expressions (such as EvalExpr) may be evaluated by `XCMDGlue` in a manner compatible with HyperTalk. Other messages and expressions (such as GetFieldByName) always assume HyperCard entities for which there are no counterpart in Director.

Appendix B

Using XObjects

This appendix describes XObjects and how to use them.

Why use XObjects

XObjects—software modules that interact with external objects—extend Director's ability by letting you interact with external software and hardware.

XObjects can be used for:

File IO	Reading and writing text files
Device control	Using NuBus cards, CD-ROMs, and videodisc and videotape players
Serial port control	Sending and receiving information through the serial port
Specialized need	Complex math, managing memory, and color palette control

If you're already familiar with using XObjects and need specific information about particular methods, see the section "XObject reference," later in this appendix.

General object theory

To understand XObjects better, it helps to have basic knowledge of objects in general. You can think of an object almost like a "black box" that has an input and an output. When you put data into the box, it performs a specific function on that data and sends the result back out of the box. Similarly, we can define an object as an independent piece of code that:

- Receives information from an outside source
- Internally evaluates or processes the information
- Returns information or performs an activity based on the result.

The benefit of this approach is that you can build reusable sections of code that have a modular structure. When you send messages to an object, you should always obtain predictable results. This saves you from having to duplicate large sections of code, and it means you can plug new “modules” (or objects) into the application at any time to add more functions and capabilities. You can also create several copies of one object to do different jobs in different circumstances.

Objects are a difficult concept at first. If the above description isn’t clear to you, don’t worry. Just try working with the examples below. Some practical experience helps you understand the theory behind XObjects.

What an XObject is

An XObject is a code resource (XCOD) that can be accessed through Lingo.The easiest way to learn about XObjects is by working with them in the message window.

To see which XObjects are available in your Director application:

1. Open the message window.
2. Type `showXlib`.
3. Press Return.
   The following list appears:

```
-- XLibraries:
-- "*Standard.xlib"
-- XObject: SerialPort      Id:200
-- XObject: FileIO          Id:1020
-- XObject: XCMDGlue        Id:2020
```

Each object in the list does the following:

- `SerialPort` sends/receives information through the serial port.
- `FileIO` reads/writes text files.
- `XCMDGlue` lets Director call standard HyperCard XCMDs and XFCNs.

These XObjects are contained in the file `*Standard.xlib`, which is part of Director. XObjects can also be contained in external files that are called from Director using the `openXlib` command. The XObject's resources can also be copied right into a movie file using utilities such as ResEdit, which makes the `openXlib` command unnecessary.

XObjects are made up of methods. In a very general sense, a method is functionally similar to a handler, except that you can't see the scripts within a method. You can list the methods within an XObject by using the message window.

To display the methods in an XObject:

▶ Type the name of the XObject followed by `(mDescribe)`, and then press Return.
The method `mDescribe` is then called, returning a descriptive list of the methods built into that particular XObject.

For example, these are the methods for the `SerialPort` XObject. Note that the characters in the left column indicate whether the method uses integers (`I`), strings (`S`), or nothing (`X`) as arguments. The "`>`" symbol in the comment shows what information the method returns back to Lingo:

```
SerialPort (mDescribe)
-- Factory: SerialPort ID:200
-- SerialPort, Tool, Version 1.1, 9/24/90
-- © 1989, 1990 Macromedia, Inc.
-- by John Thompson and Jeff Tanner.
II mNew, port -- Creates an instance of the XObject.
X mDispose -- Disposes of the XObject.
I mGetPortNum --> The port.
```

```
ISmWriteString, string -- Writes out a ¬
string of chars.
IImWriteChar, charNum -- Writes a single character.
S mReadString --> The contents of the input buffer.
I mReadChar --> A single character.
I mReadCount --> The number of characters ¬
in the input buffer.
X mReadFlush -- Clears out all the input characters.
IIImConfigChan, driverNum, serConfig
IIIImHShakeChan, driverNum, CTSenable, CTScharNum
IIIImSetUp, baudRate, stopBit, parityBit
```

Thanks to the method `mDescribe`, we can see that the `SerialPort` XObject contains methods for reading and writing strings or characters through the serial ports. There are 12 methods in the `SerialPort` XObject. For a thorough description of these methods, see the section "XObject reference" later in this appendix.

Learning how to use XObjects

You can create, use, and dispose of XObjects in the message window. This is a good place to learn basic XObject usage.

To create an instance of the `SerialPort` XObject:

▶ Type `put SerialPort (mNew, 1) into PortObject` in the message window, and then press Return.
By passing the argument 1, you create a new instance or occurrence of the `SerialPort` XObject for the printer port and place it into the variable PortObject.

To verify that the object has been successfully created:

▶ Type `put PortObject`, and then press Return.
The object appears in the message window. The display looks like this: `<Object:1e5c38>`

The numbers and letters following the word `Object:` relate to the address of the object in memory. These change with every instance, since each new instance is unique.

To verify which port is currently assigned to the object by using the method `mGetPortNum`:

▶ Type `put PortObject(mGetPortNum)`, and then press Return.
The object returns an integer (1) that corresponds to the active port.

Finally, to dispose of the instance:

▶ Type `PortObject (mDispose)` in the message window, and then press Return.

To verify that the object has been removed from memory:

▶ Type `put PortObject`, and then press Return.
If the object is no longer present, Lingo returns 0.

Basic XObject scripting

There are three basic stages in the life of an XObject:

1. Creating a new instance of the object in memory using **`mNew`**
2. Sending messages and getting results back from the object called by the methods
3. Removing the object from memory using **`mDispose`**.

In this section you learn how to perform these steps by working with the `SerialPort` XObject.

This a basic script using the `SerialPort` XObject:

```
on useSerialPort
  -- Variable "thePort" contains instance
  -- of the XObject.
  -- Use 0 for modem port, 1 for printer port.
  put SerialPort (mNew, 0) into thePort
  -- Method mwriteString sends a string
  -- (in this case, the word "howdy!")
  -- out the serial port.
  thePort (mwriteString, "howdy!")
  -- Method mreadChar reads a character in from
  -- the serial port.
  put thePort (mreadChar) into returnedChar
  -- Remove the instance of the object from memory.
  thePort (mDispose)
end useSerialPort
```

What happened here?

First, to use an XObject you have to create an instance of the object in memory. This means taking a copy of the code and placing it into RAM where you can work with it.

Then you can call methods (functions) contained in the object. The easiest way to find out the methods that are available for a particular object is to use the message window.

When you're finished with the object, dispose of the instance. Disposal of objects is very important for consistent behavior.

Working with multiple objects

A powerful feature of XObjects is their ability to make multiple instances of an object in memory. For example, imagine you wanted to control Director movies on two computers from a Director movie on one central computer. You might do this as a backup strategy in live situations or for use as a secondary computer to control video devices or play full-screen digital video). Using a null modem cable, you could connect the printer port from your "control" computer to the printer port on a "receiver" computer, and connect the modem port from the control PC to the modem port on a second "receiver." Then you could write a script for the control PC that creates two instances of the `SerialPort` XObj, one for each port, and send characters back and forth to jump through the presentation.

In the following handler, Lingo's `objectP` function is used to test for previous instances of an object:

```
on makeTwoSerialObjects
  -- Put objects into global variables so they can
  -- be called from any script. The "g" in the
  -- variable name is to remind you it's a global.
  global gModemObj, gPrinterObj
  -- objectP checks if a previous instance exists;
  -- if so, the old instance is disposed.
  if objectP (gModemObj) then gModemObj (mDispose)
  if objectP (gPrinterObj) then gPrinterObj¬
   (mDispose)
  -- Create two separate objects, one for each port,
  -- from the same XObject.
  put SerialPort (mNew, 0) into gModemObj
  put SerialPort (mNew, 1) into gPrinterObj
  -- Adding NOT to the objectP statement
  -- double-checks that the objects have been
  -- created properly.
  if not, objectP (gModemObj) then gModemObj¬
  (mDispose)
  if not objectP (gPrinterObj) then gPrinterObj¬
  (mDispose)
end makeTwoSerialObjects
```

The following handler on the “control” computer could be called from a button and would advance the presentation to the next marker by sending strings out of the serial port. The “receiver” computers would have scripts that read strings coming into the serial port, and advance the presentation accordingly. This assumes that the `SerialPort` objects have already been created on all of the computers:

```
on sendAdvanceCommand
  global gModemObj, gPrinterObj
  gModemObj (mWriteString, "next")
  gPrinterObj (mWriteString, "next")
end sendAdvanceCommand
```

This handler on the “receiver” PC connected to the modem monitors the incoming port. In actual practice, this should be called from an `on idle` handler in a movie script:

```
on receiveAdvanceCommand
  global gModemObj
  if gModemObj (mReadString) = "next" then ¬
  go to marker (1)
end receiveAdvanceCommand
```

For more information about using specific methods in the `SerialPort` XObject, see the section “XObject reference” later in this appendix.

Using basic FileIO

The `FileIO` XObject is useful for reading and writing text files. Some example uses of `FileIO` include saving user responses, names and addresses, or interactive game information. `FileIO` is also good for presentations that include text that changes frequently, because you can update the text files instead of the movies. Using multiple instances is very helpful when using the `FileIO` XObject, because you can read from and write to multiple files from multiple objects.

Syntax for FileIO

The `FileIO` XObject has three main functions:

- Write, which writes text to a file
- Read, which reads text from a file
- Append, which adds text to the end of an existing file.

When you create a new instance of the `FileIO` XObject, you must also specify which of the three functions you wish to use and which file you want to access.

The standard syntax for the `FileIO` XObject is:

`put FileIO (mNew,` *function*`,` *fileName*`) into` *objectName*

Placing a question mark (`?`) in front of the function name (i.e. "`?read`", "`?write`" or "`?append`") brings up a standard file dialog, allowing the user to select a file manually.

For example, the following statement creates a new instance—`gWriteObject`—of `FileIO` that can write to a file named "Text File":

```
put FileIO(mNew,"write","Text File") ¬
into gWriteObject
```

These handlers demonstrate how to create several different instances of the `FileIO` XObject to either write to, read from, or append to a file:

```
on writeFile
  global gWriteObject
  -- Create instance for writing to "Text File."
  put FileIO(mNew,"write","Text File") into ¬
  gWriteObject
  -- Put some sample text into a variable.
  set theText = "This text was written using ¬
  the FileIO XObject"
  -- method mWriteString writes the contents
  -- of variable "theText" to the file
  gWriteObject(mWriteString, theText)
  -- Dispose of the instance.
  gWriteObject(mDispose)
end writeFile

on readFile
  global gReadObject
  -- By adding the "?" to "read", the user ¬
  -- can select a file from a standard file dialog.
  put FileIO(mNew,"?read", "Text File") ¬
  into gReadObject
  put gReadObject(mReadLine) into theText
  -- This displays the text in the message window.
  put theText
  gReadObject(mDispose)
end readFile
```

```
on appendFile
  global gAppendObject
  put FileIO(mNew,"append","Text File") ¬
  into gAppendObject
  -- Lingo's RETURN constant puts a carriage ¬
  -- return in the file.
  set newText = RETURN & "This is the new ¬
  endpoint of the file."
  gAppendObject (mWriteString, newText)
  gAppendObject (mDispose)
end appendFile
```

Resolving pathname errors

The primary error most users encounter when using `FileIO` has to do with file pathnames. If the XObject can't find the text file you want to access it generates an error.

It's a good idea to use the Lingo `pathname` command to avoid this problem. The `pathname` function returns the full path to the current movie. When you use the `pathname` function as part of your `FileIO` script and place the text file in the same folder as the movie, Director can find the file. This handler is an example:

```
on readFile
  global gReadObject
  set gReadObject = FileIO(mNew,"read",the ¬
  pathname & "Text File")
end readFile
```

For a complete description of the methods available within the `FileIO` XObject, see the section "XObject reference" later in this appendix.

Basic device control

Many hardware manufacturers support Macromedia's XObjects through standard protocols developed by Macromedia. This is referred to as the Ortho Protocol, and is included with Macromedia's XObject Developer's Toolkit.

These hardware-specific XObjects are created and distributed by the hardware manufacturers. The Ortho standard sets out a number of common methods that should be supported by every device-specific XObject. As a result, you can send similar commands to different XObject/device combinations, and obtain similar results.

For example, the method `mSearchToFrame` is contained in XObjects that control CD-ROMs and videodisc and videotape players. When using the proper XObject for the corresponding device, the same script can be used to tell a CD-ROM, videodisc player, or videotape player to search to a given frame.

Not every device supports all methods. To determine which methods a specific device supports, see the documentation for that device.

Note *Some XObjects used to control devices work with devices other than the one the XObjects was intended for. However, the most reliable performance typically comes from XObject device drivers developed and tested by hardware manufacturers. When controlling an external device from your computer, make sure you use the driver, cables, and communications settings recommended by the manufacturer.*

The following sections use the `AppleCD XObj` to play music from an Apple CD-ROM player as an example of how to create, send messages to, and dispose of a device control XObject. Notice the use of an `on startMovie` handler to create an instance of the XObject.

Creating an instance of the XObject

This `on startMovie` handler creates an instance of the XObject:

```
on startMovie
  -- Declare a global variable that will
  -- contain the object.
  global gObjectName
  -- Create the object in active RAM.
  -- Use "pathname" command to reduce
  -- possible path errors.
  openxlib (the pathname & "AppleCD XObj")
  -- "objectP" tests for previous instances;
  -- if any exist, dispose of them.
  if objectP(gObjectName) then ¬
  gObjectName(mDispose)
  -- Place an instance of the object into
  -- variable "gObjectName."
   put AppleCD(mNew) into gObjectName
end
```

Sending commands to the device

This handler uses the method `mGetFirstFrame` to return the first frame of a specified track on a disc. In this example, you get the starting frame of track 3, and then go to and play that frame:

```
on playCD
  global gObjectName
  put gObjectName(mGetFirstFrame, 3) into SongThree
  -- SongThree is the first frame # in the 3rd track
  gObjectName(mSearchTo, SongThree)
  gObjectName(mPlay)
end playCD
```

This handler stops the device. It can either be called at a frame or from a button:

```
on stopCD
  global gObjectName
  gObjectName(mStop)
end stopCD
```

Disposing of the object

This handler disposes of the object when the movie stops. This is highly recommended; otherwise multiple instances of the object could be created and eventually cause memory errors.

```
on stopmovie
  if objectP(gObjectName) then ¬
  gObjectName(mDispose)
end stopmovie
```

Special considerations for device control

There are some special considerations you should pay attention to when working with external devices.

Because playback from a device happens over time, and because devices or communication links can fail, it is a good idea to monitor the status of the device while it executes certain functions (like searching and ejecting media). This assures that commands are properly completed and that new commands are sent at the proper time. For example, if you want to search to a frame and then begin playback, you want to be sure that the search is completed and you've arrived at the proper frame before sending a command to begin playing.

The Ortho protocol contains a special method, `mService`, which is used to return the status of a device. While the operation is still in progress, `mService` returns positive integer values. When an operation is completed successfully, `mService` returns a value of 0. If an error occurs and the device cannot complete the operation, `mService` returns a negative integer value. This error code can be used to evaluate the source of the problem. A listing of the codes that are returned by `mService` is in the section "XObject reference" later in this appendix.

The following is a sample handler that can be used to repeatedly call `mService`:

```
on awaitCompletion
  global gObjectName
  -- gDevice contains instance of an XObject
  -- for a specific device.
  -- gDeviceStatus will contain values returned by
  -- mService.By declaring gDeviceStatus global,
  -- you can share its value with other handlers
  -- that could, for example, put errors into the
  -- message window, or display them in an
  -- "alert" dialog.
  global gDevice, gDeviceStatus
  -- This repeat loop will call mService
  -- over and over again.
  -- Theoretically, this loop could
  -- repeat forever; however,
  -- you will use EXIT to jump out of the loop
  -- when mService returns a value of
  -- 0 or less indicating either that the
  -- operation was successful, or that
  -- there was an error.
  repeat while TRUE
    -- Place the value returned by mService
    -- into gDeviceStatus.
    set gDeviceStatus = gDevice (mService)
    -- Test if mService has returned 0
    -- (successfully completed)
    -- or a negative number (an error).
    -- If neither is the case, then
    -- mService must have returned a positive number
    -- indicating the operation is
    -- still in progress, and we should
    -- remain in the loop.
    if gDeviceStatus <= 0 then
    -- Jump out of the loop
    exit repeat
    end if
  end repeat
  -- Now check to see if Director exited the repeat
loop
  -- due to successful completion or
  -- an error (negative value). This tests
  -- only for negative values. If the value
  -- isn't negative, the only remaining
  -- possibility is 0 (successful), and we will
  -- automatically be returned to the script
  -- that called this one. First check a
  -- value of -1, which is a device-specific error.
  if gDeviceStatus = -1 then
  -- Put an alert dialog with customized
  -- message on screen.
  -- Method mExplain may be used to retrieve a
  -- string from the device indicating the
  -- source of the problem.
    alert "Problem with the device: " & ¬
    gDevice (mExplain)
  -- Now test for the remaining standard error codes.
  else if gDeviceStatus < 0 then
    -- Put up an alert dialog with the error code.
    alert "Problem with the device: Error # " & ¬
    gDeviceStatus
  end if
end
```

Since this handler may look a bit daunting with all the comments, the following is the handler's simplest form to help clarify what's going on:

```
on awaitCompletion
  global gDevice, gDeviceStatus
  repeat while TRUE
    set gDeviceStatus = gDevice (mService)
    if gDeviceStatus <= 0 then exit repeat
  end repeat
  if gDeviceStatus = -1 then
    alert "Problem with the device: " & ¬
    gDevice (mExplain)
  else if gDeviceStatus < 0 then
    alert "Problem with the device: Error #" &
    gDeviceStatus
  end if
end
```

To put the example in context, the following handler provides an example of how to call the `awaitCompletion` handler after initiating a command:

```
on determineCurrentFrame
  global gDevice
  -- Method mReadPos will read the current position,
  -- and method mGetValue will return the value.
  gDevice (mReadPos)
  awaitCompletion
  put gDevice (mGetValue) into currentFrame
  -- Display the current frame number
  -- in a text cast member.
  -- Using "string" function converts
  -- integer values into a text string
  -- for proper display in a text field.
  set the text of cast "frameCounter" to ¬
  string (currentFrame)
end
```

Only certain methods need to be followed by repeated calls to `mService`. For specific information on which methods require monitoring, see the section “XObject reference,” later in this appendix.

Using serial devices

Some devices, specifically videodisc and videotape players, communicate through the serial port. In these cases, you must go through two steps:

- First create an instance of the `SerialPort` XObject to communicate with the device.
- Second, use the method `msetSerialPort` (which should be contained in the device-specific XObject) to connect the device to the port. For example, the following handler does this for a Sony laserdisc player:

```
on setupSonyLaserdisc
  -- Declare global variables for
  -- serial port and Laserdisc
  global gLaserdiscObject, gPortObject
  -- Create a new instance of the SerialPort XObject.
  -- Use 0 for modem; use 1 for printer
  put SerialPort (mNew, 1) into gPortObject
  -- Open the external file containing
  -- the Sony XObject.
  -- "the pathname" command is recommended
  -- to reduce path errors.
  openXlib the pathname & "Sony XObj"
  -- Create a new instance of the Sony XObj.
  put Sony_videoDisc (mNew) into gLaserdiscObject
  -- Connect the port to gLaserdiscObject.
  gLaserdiscObject (msetSerialPort, gPortObject)
end
```

Retaining interactive control

In many cases, you may want to play a section of video or audio from a device. The Ortho protocol includes a set of methods called segment methods specifically for this purpose. To play a segment, use the following methods in this order:

1. **`mSetInPoint`**, which sets the starting point of the segment
2. **`mSetOutPoint`**, which sets the ending point of the segment
3. **`mPlayCue`**, which cues up the starting point of the segment
4. **`mPlaySegment`**, which plays the segment.

Here's an example that uses these methods to play a segment from a Pioneer laserdisc player. The first handler is an `on startMovie` handler that initializes the device:

```
on startMovie
  global gPortObject, gLaserdiscObject
  if objectP (gPortObject) then ¬
  gPortObject (mDispose)
  if objectP (gLaserdiscObject) then ¬
   gLaserdiscObject(mDispose)
  openxlib the pathname & "Pioneer XObj"
  put SerialPort (mNew,0) into gPortObject
  put Pioneer_videodisc (mNew) into gLaserdiscObject
  gLaserdiscObject (mSetSerialPort, gPortObject)
end startMovie
```

This handler defines and plays a segment of video. (If you haven't already, see the section "Special considerations for device control" earlier in this chapter for an explanation of using the `mService` method.):

```
on playVideoClip
  global gLaserdiscObject, gDeviceStatus
  -- Assume start frame number is 2000,
  -- end frame is 2500.
  gLaserdiscObject (mSetInPoint, 2000)
  gLaserdiscObject (mSetOutPoint, 2500)
  -- Cue the device, and call mService
  gLaserdiscObject (mPlayCue)
  awaitCompletion
  -- Test that mService didn't return an error,
  -- and that mPlayCue was successful.
  if gDeviceStatus >= 0 then
    -- Play the segment.
    gLaserdiscObject (mPlaySegment)
    awaitCompletion
  end if
end
```

A major consequence of `mPlaySegment` is that it removes interactive control during the playback of a segment. In effect, `mPlaySegment` is like a repeat loop that is closed to outside events until the segment has finished playing. In certain cases this could be a feature, providing a way to "lock" the user into seeing or hearing an entire segment without interruption.

In other cases, however, you may want to maintain interactivity. You can do this using individual device methods rather than segment methods. The solution is to use a `go to the frame` statement in a score script. This has the playback head loop on one frame while also testing whether the device has reached a specific ending position. Director can then capture mouse and key events every time it enters that frame.

This is a brief summary of the appropriate methods:

- Use `mSearchTo` to find a start position.
- Use `mPlay` to begin playback.
- Use `mReadPos` and `mGetValue` to test the current position.
- When the current position is less than desired ending position, then use `go to the frame`.

This sequence of events must take place over at least two frames in the score. The first frame is used to find the starting position and to initiate playback. The second frame is used to test for completion of the segment. The following is a specific example in which the start position is 2000 and the end position is 2500.

This handler for the first frame can be placed in a score script or could be in a handler (as below) called from a score script or script of a cast member:

```
on startPlayback
  global gDevice
  gDevice (mSearchTo, 2000)
  awaitCompletion
  gDevice (mPlay)
end
```

These statements test the position in the second frame:

```
-- Get the current position.
gDevice (mReadPos)
awaitCompletion
put gDevice (mGetValue) into currentPosition
-- Test the current position.
if currentPosition < 2500 then
  go to the frame
else
  --Method mStill pauses at the current position.
  gDevice (mStill)
end if
```

Testing the position of a device

Notice that the above handler uses “<” and “>” conventions rather than “=” to test the position of a device. When a device is in motion (playing, rewinding, etc.) it’s best to avoid using “=” since this tests for an exact value, and it is possible that the device will be at a slightly different position at the precise moment its position is measured. Using “<” or “>” (or alternatively “<=” and “>=”) assures that Lingo detects when the device has reached a certain location.

General tips on XObject usage

The following are some important general tips about using XObjects.

Storing instances in global variables

Instances of XObjects should almost always be contained in global, rather than local, variables. Using global variables allows you to access the object from any script, so long as you declare the global in that script. The only circumstance where you could use a local variable for an XObject would be if you were creating, using, and disposing of the object all within the same script, such as when you read in the contents of a file with the `fileIO` XObject.

Handling XObject names and filenames

When using XObjects stored in external files, the name of the XObject is often different from the name of the file which contains the object. For example, the XObject for controlling Sony videodisc players is contained in a file named "Sony XObj," but the name of the object itself is "Sony_videoDisc." To see the proper name for calling an XObject, open the message window and type `showxlib`. All the currently available XObjects will be listed with their proper names.

Handling pathnames

When using objects contained in external files, it's a good idea to keep the file in the same folder as the Director movie that contains the script for opening and creating an instance of the object. Then you can take advantage of Lingo's `pathname` command to automatically return the proper path to the current folder. This greatly reduces the potential for pathname errors when accessing the external file.

Always use mDispose

Because you can make multiple objects, it's very important that you dispose of instances as a last step and that you check for previous instances before creating new ones. Otherwise, you could have two instances in two different states using the same variable name, and Director would have no way of knowing which one to use. You can also run out of memory if your script churns new objects into RAM but never disposes of old ones. As a safety measure, use `objectP` and `mDispose` in both the `on stopMovie` and `on startMovie` handlers. See the section "Using basic device control" earlier in this appendix for an example.

You can save XObjects in a file on your computer.

Note *Never dispose of an XObject twice in the same handler.*

XObject reference

This section lists the standard methods for `SerialPort`, `FileIO`, and `OrthoPlay` XObjects.

SerialPort XObject

Use the `SerialPort` XObject to send and receive data over the Macintosh's two standard serial ports (commonly called the modem and printer ports). This XObject is built into Macromedia Director, so you don't have to open an XLibrary to use it.

The following methods are supported by this XObject. Some of them return a result code. A result code of `0` indicates success, while a negative result code indicates that an error occurred.

mNew, portNumber --> object (or errorCode)

The `mNew` method creates and returns an instance of the `SerialPort` XObject.

The *portNumber* argument can be 0 to access the modem port or 1 to access the printer port.

Example:

```
put SerialPort(mNew, 0) into port
```

mDispose

The `mDispose` method closes the serial port and disposes of the XObject instance.

Example:

```
if objectP(poprt) then port(mDispose)
```

mGetPortNum --> portNumber

The `mGetPortNum` method returns the port number for the XObject (0 for the modem port, 1 for the printer port).

Example:

```
if port(mGetPortNum) = 0 then doModem
     else doPrinter
```

mWriteString, string --> resultCode

The `mWriteString` method writes the specified string of characters to the port. (Strings are limited to 256 characters. To write a string longer than 256 characters, use a repeat loop.)

Example:

```
port(mWriteString, "Here we go")
```

mWriteChar, characterNumber --> resultCode

The `mWriteChar` method writes a single character, specified by its ACSII code number, to the port.

Example:

```
port(mWriteChar, charToNum("$"))
```

mReadString --> string

The `mReadString` method reads the contents of the port's input buffer and returns it as a string.

Example:

```
put port(mReadString) into input
```

mReadChar --> characterNumber

The `mReadChar` method reads a single character from the port's input buffer and returns its ASCII code number.

Example:

```
if port(mReadChar) = charToNum(RETURN) then doIt
```

mReadCount --> number

The `mReadCount` method returns the number of characters in the port's input buffer.

Example:

```
if port(mReadCount) < 10 then waitAWhile
```

mReadFlush

The `mReadFlush` method clears the port's input buffer.

Example:

```
if finished then port(mReadFlush)
```

mConfigChan, driverNumber, configuration -> resultCode

The `mConfigChan` method performs low-level configuration of the port. It allows the input and output sides of the port to be independently configured. (The `mSetUp` method is usually used instead of `mConfigChan`. Using `mconfigChan` provides finer control when configuring a serial port.)

The *driverNumber* argument can be 0 for the output driver or 1 for the input driver.

The *configuration* argument is the sum of four values: one for the baud rate, one for the number of stop bits, one for the parity, and one for the number of data bits. The values are shown below:

Baud rate	Value to use
300	380
600	189
1200	94
1800	62
2400	46
3600	30
4800	22
7200	14

Baud rate	Value to use
9600	10
19200	4
38400	1
57600	0

Stop bits	Value to use
1	16384
1.5	-32768
2	-16384

Parity	Value to use
None	0
Odd	4096
Even	12288

Data bits	Value to use
5	0
6	2048
7	1024
8	3072

Example:

The following statements configure both the input and the output sides of the port for 4800 baud, 1 stop bit, no parity, and 8 data bits:

```
put 22 + 16384 + 0 + 3072 into config
port(mConfigChan, 0, config)
port(mConfigChan, 1, config)
```

`mHShakeChan`, `driverNumber`, `setFlags`, `xOnChar` --> `resultCode`

The `mHShakeChan` method determines the handshaking methods used by the port. It allows the input and output sides of the port to be independently configured.

The *driverNumber* argument can be 0 for the output driver or 1 for the input driver.

The *setFlags* argument determines the handshaking methods to be used. Add the following values together for the desired methods:

- XOn/XOff output flow control: `1`
- CTS hardware handshaking: `2`
- XOn/XOff input flow control: `4`
- DTR input flow control: `8`

The *xOnChar* argument is the ASCII code number of the XOn character for XOn flow control.

Example:

The following statement enables CTS hardware handshaking, XOn/XOff input flow control (using Control-Q, ASCII code 17, for the XOn character), and DTR input flow control for the input driver.

```
port(mHShakeChan, 0, 2 + 4 + 8, 17)
```

`mSetUp`, `baudRate`, `stopBit`, `parityBit` --> `resultCode`

The `mSetup` method resets and configures both the input and the output drivers of the port.

The *baudRate* argument can be 1200, 2400, 4800, 4800, 9600, 19200, or 38400.

The *stopBit* argument can be 10 for 1 stop bit, 15 for 1.5 stop bit, or 20 for 2 stop bits.

The *parityBit* argument can be 0 for noParity, 1 for oddParity, or 2 for evenParity.

This method also configures the port for 8 data bits during asynchronous communication. No handshaking options are assigned.

Example:

The following statement configures the port for 4800 baud, 1 stop bit, no parity bit, 8 data bits, and no handshaking:

```
port(mSetUp, 4800, 10, 0)
```

Note *The `mSetup` method is the recommended way to configure a serial port. For finer control, use mConfigChan and mHShakeChan.*

FileIO XObject

You can use the `FileIO` XObject to read and write text files. This XObject is built into Director, so you don't need to open an XLibrary to use it.

The following methods are supported by this XObject. Some of them return a result code. A result code of 0 indicates success, while a negative result code indicates an error condition.

mNew, option, whichFile -> object (or errorCode)

The `mNew` method opens a file and returns a file reference object for it. It takes two strings as arguments. The first is an option string, and the second is a filename or file type (depending on the option string). The six options are:

- `read`

  This option opens the file specified by *whichFile* for reading only. (Attempting to write to the file will cause an error.) When the file is in a different folder, the specification for *whichFile* must include the pathname. It is good practice to always include the pathname in case the movie is moved from that folder later.

- `write`

  This option opens the file specified by the *whichFile* argument for reading or writing. If the file already exists, the previous contents are erased. If no file currently exists, a new one is automatically created. If the file is in a different folder than the current movie, *whichFile* must be a pathname.

- `append`

  This option opens the file specified by the *whichFile* argument for reading or writing. If the file already exists, the previous contents are left undisturbed and the current file position is set to the end of the file. Subsequent writes will add characters at the end of the file. If the file is in a different folder than the current movie, *whichFile* must be a pathname.

- `?read`

  This option puts up the standard file dialog to let the user select a file to open. Only files whose file types are specified in the *whichFile* argument (for example, “TEXT”) appear in the dialog. After the user has made a selection, the file is opened as with the `read` option.

- `?write`

  This option puts up the standard file dialog to let the user specify a file to write. The *whichFile* argument specifies the suggested filename. After the user has made a selection, the file is opened as with the `write` option.

- `?append`

  This option puts up the standard file dialog to let the user specify a file to open. The *whichFile* argument specifies the suggested filename. After the user has made a selection, the file is opened as with the `append` option.

Examples:

```
put FileIO(mNew, "read", "My Phone Book") ¬
into fileObj
```

Note *Be sure to dispose of the file reference object when you are finished using the file. Otherwise the file may remain open, and you may not be able to reopen it until you restart your computer.*

mDispose

The `mDispose` method closes a previously opened file and disposes of the file reference object.

Example:

```
if objectP(fileObj) then fileObj(mDispose)
```

Note *Be sure to call this method when you are finished using the file.*

mWriteChar, characterNumber -> resultCode

The `mWriteChar` method writes a single character, specified by its ASCII code number, to the file. The character is written at the current position in the file.

Examples:

```
fileObj(mWriteChar, charToNum("A"))
```

mWriteString, string --> resultCode

The `mWriteString` method writes the specified string of characters to the file. The string is written beginning at the current position in the file.

Examples:

```
fileObj(mWriteString, ¬
"XObjects are cool!" & RETURN)
```

Note *When you want to end a line, make sure your string ends with a Return character.*

mReadChar --> characterNumber

The `mReadChar` method reads the next character of the file and returns its ASCII code number.

Example:

```
if fileObj(mReadChar) = charToNum("?") then query
```

Note *Reading past the end of the file returns -1.*

`mReadLine --> string`

The `mReadLine` method reads the next line of the file (that is, up to and including the next Return character) and returns it as a string. The returned string ends with the Return character (except perhaps at the end of the file).

Example:

```
put fileObj(mReadLine) into nextLine
```

Note *Reading past the end of the file returns the empty string* "".

`mReadWord --> string`

The `mReadWord` method reads the next word of the file and returns it as a string. Words are delimited by spaces and Return characters. Spaces are not returned. The Return character at the end of a line is returned as a word by itself.

Example:

```
if fileObj(mReadWord) = "Macromedia" then beep
```

Note *Reading past the end of the file returns the empty string* "".

`mReadToken, skipString, breakString --> string`

The `mReadToken` method reads forward in the file, first skipping over any characters that appear in *skipString* and then saving characters until it encounters one that appears in *breakString*. The saved characters are returned as a string. If *skipString* equals the empty string "", then the returned string includes the character that caused the break.

Examples:

```
put fileObj(mReadToken, "", RETURN) into nextLine
-- same as mReadLine
put fileObj(mReadToken, " ", " " & RETURN) into
nextWord
-- same as mReadWord
```

Note *Reading past the end of the file returns the empty string* "".

mGetPosition -> integer

The `mGetPosition` method returns the position in the file where the next character will be read or written. The first position is 0.

Example:

```
put fileObj(mGetPosition) into mark
```

mSetPosition, integer

The `mSetPosition` method sets the position in the file where the next character will be read or written. The first position is 0.

Example:

```
fileObj(mSetPosition, 0)
-- move to beginning of file
```

mGetLength --> integer

The `mGetLength` method returns the number of characters in the file. The count includes spaces, tabs, Return characters, and other invisible characters.

Example:

```
fileObj(mSetPosition, fileObj(mGetLength))
-- move to end of file
```

mFileName --> string

The `mFileName` method returns the name of the file as a string.

Example:

```
if fileObj(mFileName) = "System" then beep
```

mDelete --> resultCode

The `mDelete` method deletes the file from the disk and disposes of the file reference object.

Example:

```
  fileObj(mDelete)
  if the result < 0 then ¬
  alert "That file could not be deleted."
```

Note *You do not need to call mDispose after using mDelete.*

OrthoPlay XObjects

Macromedia provides several XObjects for controlling some common devices that play video and audio source material:

- The Sony videodisc and Pioneer videodisc XObjects control many models manufactured by two popular brands of videodisc players.
- The VISCA XObject controls a wide variety of videotape recorders that use the VISCA protocol.
- The AppleCD XObject controls the CD audio capabilities of the AppleCD SC.

The methods of all these XObjects conform to a Macromedia–developed protocol known as OrthoPlay. Each XObject implements only the OrthoPlay methods that are appropriate for it. However, when two XObjects do implement the same method, it makes both devices behave similarly. As a result, you can write “generic” Lingo scripts that play source material without concerning yourself with the details of the playback device or the recorded media.

The complete set of methods defined by this protocol are described in the following sections. Some of them return a result code. A result code of `0` indicates success; a negative result code indicates an error.

Initialization and selection methods

These methods create instances of the XObjects, initialize them, and select which device they will control.

mNew --> object (or errorCode)

The `mNew` method creates and returns a new instance of the XObject.

mSetSerialPort, portObject --> resultCode

The `mSetSerialPort` method assigns an instance of the `SerialPort` XObject, which is built into Director, for the OrthoPlay XObject to use. This method should be called after `mNew` and before any other methods.

mSetInitViaDlog, initTitle --> resultCode

The `mSetInitViaDlog` method presents an initialization dialog to select the device that the XObject will control. The *initTitle* string is displayed in the dialog.

mGetInitInfo --> initString

The `mGetInitInfo` method returns initialization information as a string. The info string may be passed in the `mSetInitInfo` method to initialize another instance of the XObject to the same device.

mSetInitInfo, initString --> resultCode

The `mSetInitInfo` method initializes the XObject to the settings specified by *initString*. The argument *initString* is a string previously returned by the method `mGetInitInfo`.

mGetMaxDevices --> number

The `mGetMaxDevices` method returns the maximum number of devices that can be controlled by the XObject. Devices are numbered beginning with 1.

The methods `mGetMaxDevices`, `mGetDeviceTitle`, `mSetDevice`, `mSelectDevice`, and `mGetDevice` are implemented by XObjects that can control a set of devices. The caller can query for the devices available using the `mGetMaxDevices` and `mGetDeviceTitle`, and then assign a device with either `mSetDevice` or `mSelectDevice`.

mGetDeviceTitle, deviceNumber —> deviceTitle

The `mGetDeviceTitle` method returns the title for the specified device as a string. This title can be used to present a selection menu to the user. The integer argument *deviceNumber* is in the range from 1 to the value returned by `mGetMaxDevices`.

mSetDevice, deviceNumber —> resultCode

The `mSetDevice` method assigns the device that will be controlled by the XObject. The integer argument *deviceNumber* is in the range from 1 to the value returned by `mGetMaxDevices`.

This method is called only once, before any device methods. This method is implemented by XObjects that can control one of a set of devices from a single instance. Once a device is selected, all method calls that follow affect the selected device and only the selected device.

`mSelectDevice, deviceNumber —> resultCode`

The `mSelectDevice` method reassigns the device that will be controlled by the XObject. The integer argument *deviceNumber* is in the range from 1 to the value returned by `mGetMaxDevices`.

This method may be called more than once to switch control to different devices. This method is implemented by XObjects that can control multiple devices from a single instance. Once a device is selected, all method calls that follow affect the selected device, and only the selected device.

`mGetDevice —> deviceNumber`

The `mGetDevice` method returns the number of the device assigned with `mSetInitViaDlog`, `mSetInitInfo`, `mSetDevice`, or `mSelectDevice`.

Destruction methods

This method destroys instances of XObjects.

`mDispose`

The `mDispose` method disposes of the XObject instance, freeing the memory that it uses.

Satellite methods

These methods provide error handling, error reporting, and an idle-driven tasking mechanism. They enable foreground tasks (such as animation) to operate between calls, in cases where software-only device drivers might otherwise spend too much time waiting to communicate with the device.

mService --> conditionCode

The `mService` method should be called repeatedly after calling a device method. This method returns 0 when the command is completed, a positive value if the command is still in progress, or a negative value if an error has occurred. The condition codes are listed below:

Code	Meaning
0	OK - Command completed
1	Device-specific status
2	Waiting for acknowledgement from device
3	Stopped
4	Paused
5	Playing
6	Recording
7	Playing a segment
8	Recording a segment
9	Moving forward slower than play
10	Moving reverse slower than play
11	Moving forward faster than play
12	Moving reverse faster than play
13	Fast forward (tape unthreaded/ video disabled)
14	Rewind (tape unthreaded/video disabled)
-1	Device-specific error
-2	Operation canceled
-3	Bad parameter

Code	Meaning
-4	Not enough memory
-5	`mIdle` method not called in time
-6	No response from device
-7	Unrecognized response from device
-8	Device reports negative acknowledgement
-9	Device is offline or in local mode
-10	No medium loaded
-11	No time code
-12	Time code drop out

The device-specific condition codes 1 and -1 are used when the device is in a state that is not listed. In this case the caller can call `mExplain` to get a string that can be presented to the user.

mGetValue --> number

The `mGetValue` method returns an integer value. It is used to pick up values from methods that are followed by calls to `mService`, such as `mReadStatus` and `mReadPos`.

mCancel --> resultCode

The `mCancel` method cancels the current operation. The device is left in an undefined state and should be reset by the caller. This method should be called when `mPlaySegment` or `mRecordSegment` returns an error.

mExplain --> string

The `mExplain` method returns a string explaining the current device-specific status or error condition. It can be used to get an explanation of a condition when `mService` or `mReadStatus/mGetValue` returns a device-specific condition (1 or -1).

`mIdle —> tickCount`

The `mIdle` method is an optional satellite method implemented by an XObject that requires periodic attention. When called, the XObject should do its idle task and return the minimum number of ticks (1/60th of second) that should lapse before it is called again.

Device methods

These are the methods that do the essential work of the XObject. Unless otherwise noted, all device methods must be followed by repeated calls to `mService` until it reports completion (0) or an error condition (a negative value).

`mReadStatus`

The `mReadStatus` method initiates a request for the current status of the device. It should be followed by calls to `mService` until it reports completion or error. If `mService` reports completion, the status can then be retrieved using `mGetValue`. The possible values are listed under `mService`.

`mReadPos`

The `mReadPos` method initiates a request for the current position of the device. It should be followed by calls to `mService` until it reports completion or error. If `mService` reports completion, the frame number can be retrieved using `mGetValue`.

`mSearchTo, frameNumber`

The `mSearchTo` method initiates a search to the specified frame. It should be followed by calls to `mService` until it reports completion or error. If `mService` reports completion, the device is paused at the given frame.

`mPlay`

The `mPlay` method starts the device moving forward at standard playing speed.

mStill

The `mStill` method stops the device. For videotape and videodisc devices, `mStill` leaves a still image on the screen.

mStop

The `mStop` method stops the device. For videotape devices, it retracts the tape.

mScanForward

The `mScanForward` method moves the medium forward as fast as possible while still playing video (for videotape and videodisc devices) or audio (for audio CD devices).

mScanReverse

The `mScanReverse` method moves the medium in reverse as fast as possible while still playing video (for videotape and videodisc devices) or audio (for audio CD devices).

mPlayReverse

The `mPlayReverse` method plays the medium in reverse at standard playing speed.

mFastForward

The `mFastForward` method moves the medium forward as fast as possible. This method is intended for videotape devices, which will usually disable their video while fast forwarding.

mRewind

The `mRewind` method moves the medium in reverse as fast as possible. This method is intended for videotape devices, which will usually disable their video while rewinding.

mStepForward

For a device that can increment by one frame at a time, the `mStepForward` method moves the medium forward one frame.

`mStepReverse`

The `mStepReverse` method moves the medium in reverse one frame.

`mShuttle, speed`

The `mShuttle` method moves the medium either forward or in reverse at a speed determined by the argument *speed*. Video is always enabled. The speed is in the range –7 to 7. Positive values mean forward, zero means still, and negative values mean reverse.

`mRecord`

The `mRecord` method puts the device in record mode.

`mEject`

The `mEject` method ejects the physical medium (disc, tape, and so on) from the device.

`mPrepareMedium`

The `mPrepareMedium` method is intended to set up a freshly loaded medium. For a videodisc device it might bring the disc up to speed and detect CAV or CLV addressing. For a videotape device it might reset the tape counter (if the device does not support time code) or detect the time code type (if it does). For audio CD devices it might read the track table.

`mGetFirstTrack --> trackNumber`

The `mGetFirstTrack` method returns the first track on the medium. Track numbers begin with 1. This is an immediate method that does not require `mService`.

`mGetLastTrack --> trackNumber`

The `mGetLastTrack` method returns the last track on the medium. This is an immediate method that does not require `mService`.

`mGetFirstFrame, trackNumber —> frameNumber`

The `mGetFirstFrame` method returns the first addressable frame in the specified track. If trackNumber is 0, the first frame of the medium is returned. This is an immediate method that does not require `mService`.

`mGetLastFrame, trackNumber --> frameNumber`

The `mGetLastFrame` method returns the last addressable frame in the specified track. If trackNumber is 0, the last frame of the medium is returned. This is an immediate method that does not require `mService`.

`mGetTrack --> trackNum`

The `mGetTrack` method returns the track number for the previous `mReadPos`. This is an immediate method that does not require `mService`.

`mResetCounter`

The `mResetCounter` method resets the device's tape counter. This method is intended for videotape devices that do not have absolute time code.

`mAudioEnable, channelNumber, enableFlag`

The `mAudioEnable` method enables or disables audio playback. A disabled audio channel will not be heard. An enabled audio channel will be heard when the device is in play. The argument *channelNumber* is 1 for the left channel and 2 for the right channel. The argument *enableFlag* is `TRUE` for on and `FALSE` for off.

When you use `mAudioEnable` during playback, changes in the channel's status might not be heard until playback is stopped and then resumed.

`mAudioMute, channelNumber, enableFlag`

The `mAudioMute` method enables or disables audio muting. A muted channel will only be heard at play speeds. An unmuted channel will be heard at non-play speeds. (This may cause speaker damage.) The argument *channelNumber* is 1 for the left channel and 2 for the right channel. The argument *enableFlag* is `TRUE` for on and `FALSE` for off.

mVideoEnable, channelNumber, enableFlag

The `mVideoEnable` method enables or disables video display. The argument *channelNumber* is 0 for Macintosh graphics (for video overlay), 1 for the first video input source, 2 for the second video input source, and so on. The argument *enableFlag* is `TRUE` for on and `FALSE` for off.

mShowFrame, enableFlag

The `mShowFrame` method enables or disables the display of frame numbers. The argument *enableFlag* is `TRUE` for on and `FALSE` for off. This method is intended for videodisc devices.

mGetFrameResolution --> framesPerSecond

The `mGetFrameResolution` method returns the frames-per-second rate of the device. If the method is not present, 30 frames per second is assumed. This value does not reflect the presence of drop frames. PAL devices will return 25. This is an immediate method that does not require `mService`.

mSetFrameResolution, fps --> resultCode

The `mSetFrameResolution` method is provided when the XObject cannot detect the frames-per-second rate of the device, allowing the user to set this value. Acceptable values are 30 (NTSC), 25 (PAL), 24 (Film transferred to NTSC with 3-2 pull down). This is needed for an XObject to address CLV-type discs where frames must be converted to seconds. This is an immediate method that does not require `mService`.

mHasDropFrames --> TRUE/FALSE

The `mHasDropFrames` method returns `TRUE` if the medium in the device uses SMPTE dropframes and `FALSE` otherwise. (If drop frames are used, the rate is 30 per second, skipping frames 0 and 1 every minute, with the exception of multiples of 10 minutes.) If this method is not provided, `FALSE` is assumed. This is an immediate method that does not require `mService`.

mSendRaw, string

The `mSendRaw` method sends a raw command string to the device. It is intended for testing purposes. This is an immediate method that does not require `mService`.

mReadRaw —> string

The `mReadRaw` method reads raw status back from the device. It is intended for testing purposes. This is an immediate method that does not require `mService`.

Segment support methods

These methods set instance variables used by the segment methods `mPlaySegment` and `mRecordSegment`.

mSetInPoint, inFrame --> resultCode

The `mSetInPoint` method sets the inpoint to be used for all device commands that require an inpoint. After an edit the inpoint is automatically advanced to the point following the last point of the edit, so for common sequential editing it is necessary to call this method only once. Altering the inpoint does not alter the number of frames for the next task.

mSetOutPoint, outFrame --> resultCode

The `mSetOutPoint` method sets the outpoint to be used for all device commands that require an outpoint. Video XObjects do not have a true outpoint. This method is provided for symmetry with `mSetInPoint`. It is actually an alternate method of setting a duration for programmed device methods. Any value set with this method should "float" in relation to the inpoint (so that the duration value is always preserved). Therefore, setting the inpoint should always precede setting the outpoint when using this style.

mSetDuration, nFrames --> resultCode

The `mSetDuration` method sets the duration for segment methods. Altering this value leaves the edit inpoint intact.

mGetMinDuration --> nFrames

The `mGetMinDuration` method returns the minimum number of frames possible in an edit.

mSetPreroll, nFrames --> resultCode

The `mSetPreroll` method sets the preroll length. This is the number of frames before the inpoint to cue to before a `mRecordSegment`.

mGetPreroll --> nFrames

The `mGetPreroll` method returns the preroll time in frames. For consumer-level devices preroll is taken to mean the duration from when `mRecordSegment` returns to when the device goes into record.

mSetPostroll, nFrames --> resultCode

The `mSetPostroll` method sets the post-roll length. This the number of frames after the outpoint to stop at after a `mRecordSegment`.

mGetPostroll --> nFrames

The `mGetPostroll` method returns the postroll time in frames.

mSetFieldDominance, oddEven --> resultCode

The `mSetFieldDominance` method informs the XObject of the device's field dominance hardware setting. (The field dominance setting determines the field an edit record will cut in on.) The argument *oddEven* is 1 for odd or 2 for even.

Segment methods

These methods are used for playing and recording segments:

mPlayCue

The `mPlayCue` method initiates a search to the previously set inpoint. It should be followed by calls to `mService` until it reports completion or error. If `mService` reports completion, the device is paused at the inpoint. This method may be used before `mPlaySegment`.

mPlaySegment

The `mPlaySegment` method plays to the previously set outpoint. It should be followed by calls to `mService` until it reports completion or error. If `mService` reports completion, the device is at the outpoint. The inpoint and outpoint remain unaffected. To play from the inpoint, precede this call with a call to `mPlayCue`; otherwise the device will play from the current location to the outpoint. During an `mPlaySegment`, `mGetValue` may be called to get the current position of the device.

mRecordCue

The `mRecordCue` method initiates a search to the previously set preroll point. It should be followed by calls to `mService` until it reports completion or error. If `mService` reports completion then the device is paused at the preroll point. This method must be used before `mRecordSegment`.

mRecordSegment

The `mRecordSegment` method starts the record process. It should be followed by calls to `mService` until it reports completion or error. The caller should use the `mGetPreroll` method to determine how much time will lapse from when this method returns to when the device is actually recording. The inpoint is automatically advanced once the edit is complete. If `mCancel` is called while the edit is in process, the inpoint should remain set at the position it occupied prior to the call to `mRecordCue`. The actual position of the head after `mRecordSegment` is undefined to facilitate repeated edits. After all recording is complete, the caller must put the device into the desired state. If a device supports video-only or audio-only edits, it should implement `mRecordVideoEnable` and `mRecordAudioEnable`. If these methods are not implemented it is assumed that recording will affect both video and audio. During an `mRecordSegment`, `mGetValue` may be called to get the current position of the device.

mRecordVideoEnable, enableFlag

The `mRecordVideoEnable` method enables or disables the recording of video during an `mRecordSegment`. The argument *enableFlag* is `TRUE` for on or `FALSE` for off. This method should be called before `mRecordCue`.

mRecordAudioEnable, channelNumber, enableFlag

The `mRecordAudioEnable` method enables or disables the recording of audio during an `mRecordSegment`. The argument *channelNumber* is 1 for the left channel and 2 for the right channel. The argument *enableFlag* is `TRUE` for on or `FALSE` for off. This method should be called before `mRecordCue`.

mAssembleRecord --> resultCode

The `mAssembleRecord` method is called before `mRecordCue` to enable an assemble edit. During an assemble edit video, audio, control track, and timecode are recorded onto the tape. There must be valid video and timecode during the preroll period of the edit.

mPreviewRecord --> errorCode

The `mPreviewRecord` method is called before `mRecordCue` to enable a preview edit. The recording will not take place but the edit will be simulated by switching video and/or audio at the in and out points.

mGotoInPoint

The `mGotoInPoint` method initiates a search to the previously set inpoint. It should be followed by calls to `mService` until it reports completion or error. This method is implemented by frame-accurate videotape devices as a consistency check.

mGotoOutPoint

The `mGotoOutPoint` method initiates a search to the previously set outpoint. It should be followed by calls to `mService` until it reports completion or error. This method is implemented by frame-accurate videotape devices as a consistency check.

mGotoPrerollPoint

The `mGotoPrerollPoint` method initiates a search to the previously set preroll point. It should be followed by calls to `mService` until it reports completion or error. This method is implemented by frame-accurate videotape devices as a consistency check.

mGotoPostrollPoint

The `mGotoPostrollPoint` method initiates a search to the previously set postroll point. It should be followed by calls to `mService` until it reports completion or error. This method is implemented by frame-accurate videotape devices to as a consistency check.

Appendix C

Factories

A factory is a particular kind of Lingo script in which you define your own objects. Once you've defined an object within the factory, you can subsequently call on the factory to create as many instances of the object as you want throughout a movie.

This appendix describes factories, their uses and how to create them.

Introduction to factories

Factories are useful for making more than one of an item such as bouncing balls or flying birds. Let's say, for example, that you are designing an interactive game. The game lets the user decide a level of difficulty from one to ten. A level of one means that the user will be able to zap one animated alien object. A level of ten lets the user do battle with ten animated alien objects.

If you didn't use a factory to create your aliens, you'd have to create ten different sequences in your movie, one for each level of difficulty, and place one alien sprite in the first segment, two in the second, and so on. When an alien is hit by user-fire, you want it to explode and disappear from the screen. Furthermore, you want all your aliens to move randomly around the screen and every once in awhile to fire phaser shots of their own at the user. In the ten-alien non-factory case, you'd have a lot of things to keep track of in your scripts: alien coordinates, hits or misses, how many aliens are left, and so on. Things get complicated quickly. One alternative is to use a factory.

In our example, you would write an "alien factory" for the game. In the factory, you'd define what an alien is, how it behaves, what messages it can respond to, and any other characteristics required to specify the object and its overall behavior. When the movie is running, the factory "builds" aliens in response to the user's input, as many as needed for a given game. When an alien is hit the alien object is removed from the screen and from memory. This ability to create and destroy objects dynamically, on command is an important characteristic of factories.

Furthermore, each alien object can record its own movement and whether or not it has been hit. All the aliens can respond to the same set of messages. (In our alien game, for example, you might create a `move` handler, or a `disintegrate` handler.) Each alien is called an instance of the object. Each instance of our alien can have its own data (such as its current screen position).

Note *Even though our examples in this appendix are associated with a graphic cast member that you can see on the screen, a factory object does not have to have a visual component. You can create objects that reside in memory and perform other functions, such as controlling multiple videodisc players.*

Factories, then, allow you to create more efficient, compact, and straightforward scripts. You can define as many different kinds of objects in a factory as you want.

Recall that in other Lingo scripts, you used the `on` keyword to define handlers. The handler name specifies the message to which the handler responds. In factories, you also specify the messages to which an object can respond. These are called methods, and the rules for defining a method are slightly different from those you use to define a handler.

▶ **Tip** *You can also use a factory to create and manage an array, to control a series of related objects with the same methods and functions. However, a simpler alternative is using lists to create and manage an array. For information about lists, see the section "Using Lists," in Chapter 9.*

Factories are usually written in a movie script.

Objects and messages

Factory objects communicate with other Lingo scripts through their messages. In a Lingo script, a message might be sent as follows:

```
put PioneerLaserdisc(mNew, 0) into videodisc
videodisc(mStopAtFrame, 22500)
```

The first line of this script creates an object, here called `videodisc`, by sending the message `mNew` to the `PioneerLaserdisc` factory. In this example, the `0` argument indicates that the videodisc player is connected to the modem port of the computer.

In the second line, the message `mStopAtFrame` is sent to the newly created object. The object's `mStopAtFrame` method will stop the videodisc player at a particular frame.

Assume that you attach a second videodisc player to your computer. You can create a second object using the same factory:

```
put laserDisc(mNew, 1) into secondDisc
secondDisc(mStopAtFrame, 22500)
```

At this point, you have two devices which are synchronized, and positioned at the same frame, 22500.

The syntax shown in this example is used by Lingo scripts to communicate with Lingo objects. Every object has a set of methods that it provides to Lingo scripts. In the statement `videodisc (mStopAtFrame, 22500)`, the object is `videodisc` and the first argument, `mStopAtFrame`, is the message being sent (the same as the name of the specific method being called).

In addition, each method can have a set of arguments that it expects to receive. If the method expects arguments, these follow the method name. In this case, the argument `22500` is passed to the method `mStopAtFrame` in object `videodisc`.

When you write your factory, you can specify how the objects created by the factory receive messages, using methods and their arguments. The object can receive real-time input from a variety of sources: mouse or keyboard, a sequence of predefined data from a text cast member, or input from one of the computer's serial ports.

How factories are defined

You define a factory in a movie script. The definition always begins with the `factory` keyword, followed by the factory name:

`factory` *factoryName*

▶ ***Tip*** *In previous versions of Director, factories could be created only in text cast members. The current version requires that you put them in a movie script.*

The factory name uses alphanumeric characters (no special characters or punctuation marks). A factory name can be only one word; no spaces are allowed.

The statement containing the `factory` keyword and its name is followed by a series of method definitions, defined by the `method` keyword:

`method` *messageName1* `[`*arg1*`,` *arg2*`...]`

statements

`end` *messageName1*

`method` *nextMessageName* `[`*arg1*`,` *arg2*`...]`

    statements

`end` *nextMessageName*

Note *In the preceding example, words in typewriter type are those elements that you enter exactly as shown. The words or phrases in italics are placeholders that describe the general parameter or argument for which you supply specifics. The square brackets [ ] enclose optional elements that you include if needed. (You don't type the square brackets, though.) Optional elements may or may not change what a statement does. For more about these conventions, see the Lingo Dictionary.*

Factories also make use of instance variables, defined by the `instance` keyword and discussed next.

Instance variables

In Lingo scripts outside of factories there can be two kinds of variables: global and local. Global variables remain in existence for the duration of a movie. Local variables only exist while the handler or script that created it is being executed. Factories can use local and global variables, and can include a third kind of variable: instance variables. A factory can assign instance variables to specific objects. Instance variables contain a unique set of values specific to each individual object, even though the variables have the same name. The methods of a factory use the instance variables.

An instance variable is available only to the object with which it is associated. The value of an instance variable is established when the object is created, or when a method is used to change it. Each instance variable and its value persists as long as the object itself persists.

To define an instance variable, you must use the instance keyword, otherwise the factory will assume it is a local (temporary) variable.

You would typically define all your instance variables in the `mNew` method of a factory, the method that creates new objects. Subsequently, the values of instance variables can be changed by other methods.

For example:

```
method mNew parameter1, parameter2
  instance vName1, vName2
  set vName1 = parameter1
  set vName2 = parameter2
end mNew
```

In this example, the first statement defines two parameters that will be used to pass values to two variables. The second line defines two instance variables. The third line sets the initial values for the two variables.

The perFrameHook property

The `perFrameHook` property, when assigned to an object, causes an interrupt to occur at every frame when the playback head advances. When this interrupt occurs, the specified object calls a special message called `mAtFrame`. You define in a factory what actions the `mAtFrame` method performs. This is a much simpler way of calling a script that needs to be executed every frame (`perFrameHook` is especially useful when recording to videotape frame-per-frame). For specific information about the `perFrameHook` property, see the *Lingo Dictionary*.

Creating objects from factories

After you've defined a factory, you can use it to "build" as many instances of the factory's objects as you want.

An instance of an object is created by calling the `mNew` handler of the factory. The object can then use any of the factory's handlers for sending messages and determining new values.

Objects are created with the name of the factory and the `mNew` method:

`put` *myFactory* `(mNew,` *arg1*`,` *arg2*`,...)` `into` *myObject*

Special methods in factories

Three special predefined methods are available to every object, and do not need to be defined within your factories. These are: `mPut`, `mGet`, and `mDispose`. These are described next.

Creating and using object arrays

The `mPut` method places values in an array. Every instance of a factory object automatically has an array associated with it. In fact, you might create a factory just so you can use its built-in arrays as containers. Arrays are useful for containing a "variable number of variables"— a number of values at various locations within the array.

Here is the syntax for `mPut`:

objectname`(mPut,` *n*`,` *value*`)`

The `mPut` method places value at location n in an array. The value *n* must be an integer that is equal to or greater than zero. Subsequently, you can use `mGet` to retrieve the value. *Value* can be any value, a number, a string, or another object.

`mGet` retrieves a value from the array. The syntax is:

`put` *objectname*`(mGet,` *n*`) into` *variable*

The `mGet` method returns the value at location n in an array created with `mPut`. Once again, *n* must be an integer that is equal to or greater than zero.

Removing an object from memory

Another method that is automatically available for every object is `mDispose`. Like `mPut` and `mGet`, `mDispose` does not need to be defined in your factory script before it can be used.

`mDispose` deletes an object from memory. The syntax is:

objectName`(mDispose)`

The `mDispose` method removes the object *objectName* from memory. Use `mDispose` to free up memory when an object is no longer needed.

The me keyword

Factory objects can also call their own methods by using the `me` keyword. The `me` keyword is equivalent to the name of the object whose method is being called. In the example here, the `animateBird` method is called from within the `lakeScene` factory:

```
factory lakeScene
  method mNew …
  …
end mNew
method animateBird startV, startH, speed
  …
end animateBird
…
method fly
  …
  me(animateBird, 100, 100, 2)
  …
end fly
```

The `me` keyword is useful when you want to call the same method with different objects. This way, you don't have to specify the individual object's name each time you call that method.

Index

Acknowledgements

Written by Joe Schmitz.

Appendixes A and B by Jeff Essex.

Edited by Toni Haskell.

Production by Lee Allis.

Special thanks to Mark Castle, John Dowdell, Bill Edwards, David Shields, Dan Sadowski, John Thompson, and John Ware.