20
CHAPTER

Lingo Troubleshooting

✦ ✦ ✦ ✦

In This Chapter

Tracing script actions in the Message window

Executing instructions and handlers in the Message window

Tracking variables in the Watcher window

Checking handlers by using the Debugger window

✦ ✦ ✦ ✦

Have you ever had a really bad code day? If not, you will. When that happens, Director includes a variety of built-in troubleshooting tools. Paraphrasing two common themes, it's important to realize that "bad things happen to good programmers" and "where your problem is (in the script) is where you can fix it."

Programming errors happen to everyone. Director helps by

- Alerting you to syntax errors when you close the Script window
- Providing troubleshooting tools — the Watcher window, breakpoints, and the Debugger window — to help you locate errors in logic

Troubleshooting Your Scripts

When your Lingo script does not work the way you expect it to, the first thing to do is to identify the problem. What is it that is not working the way you expected? Is a button click not playing a sound? Is the movie not branching to the correct frame? Does the movie quit unexpectedly?

If you are working on a team project, read and understand the comments inserted by the other team programmers. If you are writing Lingo script as part of a team effort, make sure you are adequately commenting your own scripts. Valuable time is wasted when you or someone else must decipher the scripts from scratch.

Troubleshooting involves three techniques that are useful when identifying and solving problems within your coding. Director provides tools to help you with each technique. The techniques are:

✦ Locating the problem

✦ Identifying syntax and spelling errors

✦ Correcting errors of logic

Each technique, discussed in the following sections, brings you closer to creating bug-free, efficient code.

Locating the problem

Sometimes finding the problem really means locating where the problem begins. The debugging tools provided by Director can assist you in locating the origin of the problem. For example, watching the frames as they are executed in the Message window can be helpful. You can also execute Lingo scripts from the Message window to learn whether the problem resides in a particular script.

Asking the following questions will help:

✦ Does the problem occur only on specific computers or only on computers with certain display settings?

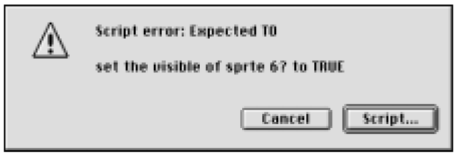
✦ Is the error occurring with a sprite and the way it displays?

✦ Is the error occurring only with certain keystrokes or combination of keystrokes?

Identifying syntax and spelling errors

The most common Lingo error occurs when you mistype a command or use the wrong syntax for a command or property. Each time you close the Script window, Director automatically looks for syntax errors or errors resulting from misspelling a word. For example, if you type the following command in a script and close the Script window, you get the error message shown in Figure 20-1.

```
set the visible of sprte 6 to TRUE
```


Script error: Expected TO

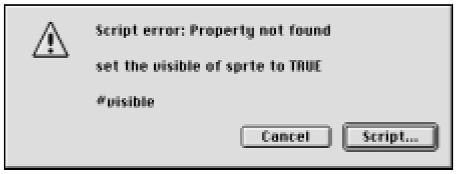
set the visible of sprte 6? to TRUE

Figure 20-1: A script error message

The error message offers Director's view of the potential problem. In this case, Director assumes *sprte* is a variable holding a reference to something, a sprite reference such as `(sprite 1)`, a child object, or perhaps a property list, which might have the visible property. That being the case, right after the variable, there should be the word `to`, but instead an integer is there. Director is confused and puts a question mark after the part that confuses it. Get rid of the 6, and try it again in the Message window, as follows:

```
set the visible of sprte to TRUE
```

We get the error notification shown in Figure 20-2.



```
Script error: Property not found
set the visible of sprte to TRUE
#visible
```

Figure 20-2: A slightly more helpful script error message

Now we can see why Director wanted the `to` instead of the `6`. It thinks our misspelling of the word *sprte* was actually a variable.

Note

Director's capabilities do not include the intelligence to recognize that a simple misspelled word is the problem, as opposed to the error of a variable's being used before it is assigned a value. When debugging a script, your job is to examine the error message and interpret it in the context of the remainder of the Lingo script.

Checking for spelling errors

When checking for syntax errors (the most frequent type of error), you should always check the spelling of commands, functions, and properties. Be sure that variable names are used consistently and that spaces and punctuation marks are included when necessary. Director can only respond to your script as you typed it, not as you intended to type it.

Checking for missing quotation marks

Another common syntax error results when you omit quotation marks. Make sure you've used the required quotation marks (" ") around the names of cast members, markers, and strings.

Checking for missing command parameters

Lingo commands, properties, and other instructions might need required parameters in order for Lingo to know how to interpret the instruction. Always check for missing parameters or arguments that are required by a command or instruction.

Checking for the correct command syntax

When you are not sure of a Lingo command's syntax and required parameters, there are three tools available to assist you: the Alphabetical Lingo button, the Categorized Lingo button, and the Director Help command.

Note

The Alphabetical Lingo and Categorized Lingo buttons are available on the Script window toolbar (see Figure 20-3) and in the Message window toolbar.

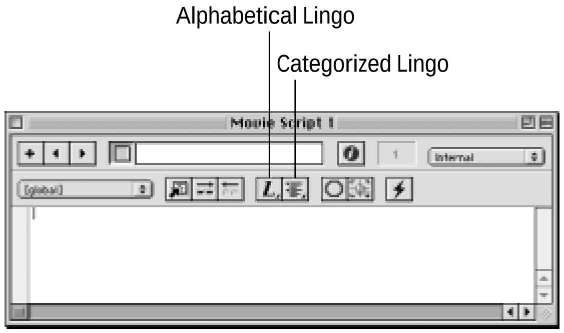


Figure 20-3: Two tools in the Script window, the Alphabetical Lingo button and the Categorized Lingo button, help you with Lingo syntax.

Place the cursor at the location in your script where you want to use the command, and click the Alphabetical Lingo button. Search the list to locate the desired command. Select it and the command's correct syntax and placeholders for required parameters are added to the Script. The command is added to the Script window at the current location of the insertion point.

The Categorized Lingo pull-down list works the same way as the Alphabetical list. Search the groups of commands (for example, Navigation, Network, User Interaction) to locate the desired command. When you locate the command and select it, the command's correct syntax and placeholders for required parameters are added to the Script. The command is added to the Script window at the current cursor location.

The third tool for fixing your Lingo syntax is Director's online help. The online help has many useful topics and the entire Lingo dictionary in electronic form.

Correcting errors of logic

Always remember the Script window's closing without displaying an Alert window is no guarantee that your Lingo script is error free. It only indicates that the syntax checker did not locate or recognize errors in your script. As a parallel, consider that a word processor's spelling checker classifies a word as misspelled if the word is not contained in the speller's dictionary file. Proper names, even when they're spelled correctly, are often flagged as misspelled. The spelling checker only highlights words it does not find in its dictionary file. Any word not found is flagged as possibly incorrect.

The syntax checker in Director operates on a similar principle. It can only locate problems that appear to violate the patterns established in Lingo's rules file. The syntax checker that works with the Script window can identify syntax errors, but it cannot identify errors of logic.

Logic errors result when you assign incorrect values to variables or parameters. The Lingo command that uses or assigns values to a variable might be absolutely correct in its syntax, yet the value stored by the variable might still be the problem. To troubleshoot errors of logic requires that you step through each line of code, in the order they are executed, and look for unexpected results that in turn play havoc with your program. This process can be very complex, because variables might store different values (or strings) depending on which events and actions occur as the movie plays.

To tackle errors of logic, you can use Director's Message window (with its set of debugging buttons), the Debugger window, and the Watcher window.

Debugging in the Message Window

The first line of defense against most script problems is to use the Message window. To open the Message window, choose Window ➪ Message, or press Command+M (Ctrl+M). You can use the Message window to observe the instructions contained in the handlers of a movie as it plays. The debugging buttons on the Message window's toolbar, identified in Figure 20-4, provide access to debugging tools. Each button, along with its function, is listed in Table 20-1. Some of these buttons are also available in the Script window and the Debugger window.

Each of these troubleshooting tools gives you access to fix-it features within Director. As you create scripts in Director, you'll probably find that the most useful tools are the Alphabetical Lingo and Categorized Lingo buttons. With both, you can type an instruction in the Message window and observe Director's reaction to the command in the Message window or on the Stage.

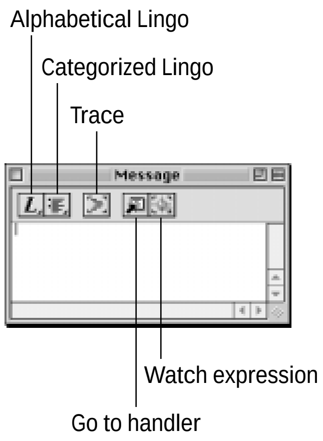


Figure 20-4: Message window's debugging buttons

Table 20-1
Lingo Troubleshooting Tools

Button Name	***Function and Purpose***	***Location***
Alphabetical Lingo	Displays an alphabetical listing of all Lingo commands. Locate the desired command on the menu and select it to insert the command and placeholders for all required parameters.	Message window and Script window
Categorized Lingo	Displays a list of all Lingo commands, sorted by function. Locate the desired command on the menu and select it to insert the command and placeholders for all required parameters.	Message window and Script window
Trace	Toggles ON/OFF the display of currently executing Lingo scripts.	Message window
Go To Handler	Opens the Script window and displays the handler referenced in the current instruction. Places the insertion point at that location in the script.	Message window and Debugger window
Watch Expression	Adds the currently selected variable or expression to the "watch list" tracked by the Watcher window.	Debugger window, Message window, and Script window

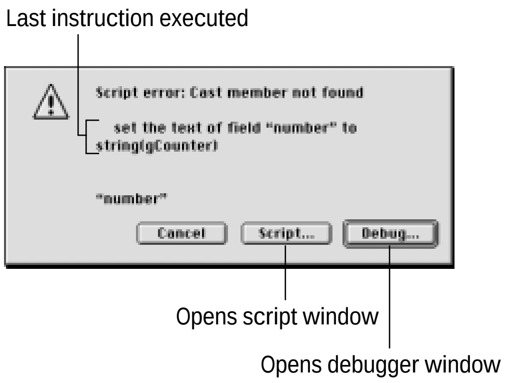
In the following steps, you use the Message window to trace the execution of commands in a movie and to locate a logic error. Using the Trace button, you can watch the execution of all instructions (in any script) as they are displayed in the Message window. In its current state, the movie includes a button (cast member 1), a field (cast member 2), two behaviors (cast members 3 and 4), and a movie script (cast member 5). The Count to 10 button has a behavior attached to it. The behavior is stored in cast member slot 4.

On the CD-ROM

You can find the file counter.dir on your companion CD-ROM in the EXERCISE:CH20 (EXERCISE\CH20) folder.

Tracing Script Actions in the Message Window

1. Open the file counter.dir in Director. Open the Cast window and examine the cast members. Then open the Message window (press Command+M or Ctrl+M).
2. Click the Trace button in the Message window, identified in Figure 20-4. This turns on the Trace function.
3. Arrange the open Cast window and Message window so that you can see the field (cast member 2) and the Count to 10 button on the Stage.
4. Use the Play button on the toolbar to play the movie. The movie stops almost immediately and displays the error message shown in Figure 20-5. In addition to the error, the Message window shows the last instruction executed before the movie stopped. And the three buttons at the bottom are for opening the Debugger window (discussed later), opening the Script window so you can correct the error, and closing the Script Error message box.



```
Script error: Cast member not found

   set the text of field "number" to
string(gCounter)

"number"
```

Figure 20-5: The “Cast member not found” script error message

5. Click the Script button. When the Script window opens, examine the script and try to identify any errors or problems. If you don't see anything obvious, don't worry. Just close the Script window.

   Director now closes the window without issuing a syntax-error warning; this indicates that the problem is not related to a syntax error in the script. The real problem is that the script refers to a cast member ("number") that does not exist.

6. Select the field cast member (slot 2) in the Cast window, click in the Name field, and type **number**. Press Enter to apply the new cast member name to the field.

7. Save the movie as **counter1.dir**.

8. Leave the Message window open and play the movie. As soon as you start the movie, `go to the frame` instructions scroll down the Message window.

9. Click the Count to 10 button, and you should notice that each instruction from script cast member 4 is displayed in the Message window after you click the button and while the movie plays.

10. Stop the movie.

11. Use the Message window scroll bar to scroll back to the point where you clicked the Count to 10 button.

Now let's examine the results. At the point where you click the Count to 10 button, the Message window displays the following information. The Message window constantly updates the value of the variable `gCounter`. At each step, you can examine the program's flow and see which handler and instructions are executed. Note the first few cycles (as the `gCounter` variable is incremented from 0 to 3):

```
== Script: (member 4 of castLib 1) Handler: mouseUp
--> gCounter = 0
== gCounter = 0
--> repeat while gCounter < 11
--> member("number").text = string(gCounter)
--> gCounter = gCounter + 1
== gCounter = 1
--> end repeat
--> repeat while gCounter < 11
--> member("number").text = string(gCounter)
--> gCounter = gCounter + 1
== gCounter = 2
--> end repeat
--> repeat while gCounter < 11
--> member("number").text = string(gCounter)
--> gCounter = gCounter + 1
== gCounter = 3
```

If you scroll back to the beginning of the Message window listings, you see text similar to this:

```
== Movie: Mac HD:desktop stuff:CH20:COUNTER1.DIR Frame: 1
Script: (member 5 of castLib 1) Handler: startMovie
--> gCounter = 0
== gCounter = 0
--> member("number").text = string(gCounter)
--> end
== Script: (member 3 of castLib 1) Handler: exitFrame
--> go to the frame
--> end
```

The Message window lists the path to the movie, the frame number, the location of the script that is executing [in the form of a member reference: `(member 5 of castLib 1)`], and then the name of the handler executing. After the `on startMovie` handler finishes (which establishes the initial value of the variable `gCounter` and places it in the number field), the cast member script 3 with an `on exitFrame` handler runs. It includes a `go to the frame` instruction.

Testing commands directly in the Message window

You can also type and test scripts in the Message window (you have already done this in earlier chapters). The Message window enables you to prototype and test specific instructions to determine if the actual results match your anticipated results.

- ✦ To test a single line of code that you suspect might be causing a problem, type it directly into the Message window. If the command works in the Message window, some other part of your program must be causing the error.
- ✦ To test a handler in the Message window, type the name of the handler and press Enter. The handler is immediately executed, and if any errors occur, an Alert message displays.

On the CD-ROM

You can find the file change.dir on your companion CD-ROM in the EXERCISE: CH20 (EXERCISE\CH20) folder.

Executing Instructions and Handlers in the Message Window

1. Open the change.dir movie.
2. In the Message window, click the Trace button to turn off the Trace feature.
3. In the Message window, type the following:

   ```
   set the visible of sprite 1 to FALSE
   ```

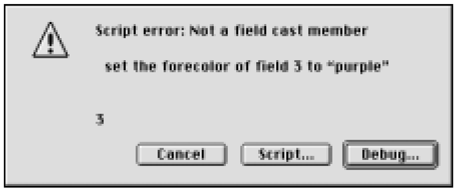
   Press the Enter key. The button (sprite 1) disappears.

4. Type the following:

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

Press the Enter key. The button reappears.

5. The movie also has another handler in cast member 50, named `on change`. In the Message window, type `change` and press Enter. The script error message shown in Figure 20-6 appears.



Script error: Not a field cast member

set the forecolor of field 3 to "purple"

3

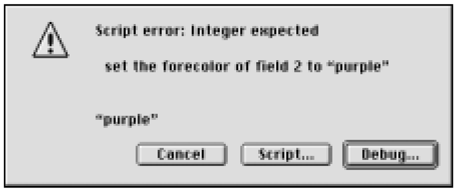
Figure 20-6: The "Not a field cast member" script error message.

6. Click the Script button in the error box. Three lines in the script refer to field 3. Because there is a mismatch between the type of cast member in slot 3 and the type of cast member specified in the command (field), you must change the instruction.

7. Change all three references in the `on change` handler from field 1 to field 2.

8. Close the Script window, and save the movie as **change1.dir**.

9. Type **change** in the Message window again and press Enter. Director displays a script error indicating that an integer (whole number) is expected, as shown in Figure 20-7. The word "`purple`" in the script is listed as the culprit. The `foreColor` property must be set by using a numeric value (not a color name) that represents the color in the current palette.



Script error: Integer expected

set the forecolor of field 2 to "purple"

"purple"

Figure 20-7: The "Integer expected" script error message

10. In the error box, click the Script button to open the Script window.

11. Delete the word `purple` and the double quotation marks that surround it. Change the instruction to read as follows:

```
set the forecolor of field 2 to 54
```

Note

The color 54 is purple when using either the System-Win or the System-Mac palette.

12. Close the Script window, and save the movie again as change1.dir.

13. In the Message window, execute the `on change` handler by typing **change** and pressing Enter. The number 0 increases in size, switches to purple, and changes to an italic font style.

Using the showGlobals command

Another useful purpose for the Message window is to display variables that are currently available as globals. This can be helpful in your debugging efforts for two reasons:

- ✦ You might expect a variable to be global and it is not (that is, you did not declare it as a global variable).
- ✦ You want to know the value currently stored by a global variable.

To determine which variables are currently declared as globals, you open the Message window and type **showGlobals**. You get a list of all current global variables, as in this example:

```
gName = "John"
gAge = 34
showGlobals

-- Global Variables --
gName = "John"
gAge = 34
version = "8.0"
```

The `version` global is always there; it holds the version of Director you are using.

Using the globals property

Director 7 introduced `the globals` property. This is a special property list of all current globals that have a value other than VOID. Where the `showGlobals` command is useful while authoring, `the globals` command can be used in your program. Every time you create a new global variable in your program (or the Message window), the global variable is added to `the globals`:

```
put the globals
-- (the globals)
put (the globals).count()
-- 3
```

```
put (the globals).getPropAt(1)
-- #gName
put (the globals).getProp(1)
-- "John"
```

Although the globals seems like a property list, it is more like a crippled property list. It only responds to these functions:

```
count()
getPropAt()
getProp()
getAProp()
```

`getProp()` functions differently from normal property lists. If you use `getProp()` to try to get a property that does not exist in a property list, you will get a syntax error. If you do that with `the globals`, `<Void>` will be returned. (See the “VOID versus <Void>” sidebar if you’re unclear on the difference.) So, when it comes to `the globals`, `getProp()` and `getAProp()` work identically.

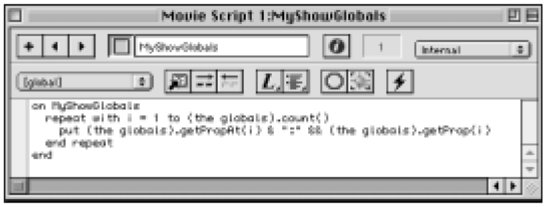
Now try the following:

```
put (the globals).count()
-- 3
clearGlobals
showGlobals

-- Global Variables --
version = "8.0"

put (the globals).count()
-- 3
```

You just cleared `the globals`, but `version` is still there because it’s always there. But what is going on with `the globals` property? There is a problem with `the globals`. If you use `clearGlobals`, it removes the values (they are set to VOID) from `the globals`, but not the properties. So, Director is still reporting three items in `the globals`, even though two of those have values of `<Void>`. This can be illustrated by creating a handler to show properties and their values, as shown in Figure 20-8.



```
on MyShowGlobals
  repeat with i = 1 to (the globals).count()
    put (the globals).getPropAt(i) & ":" && (the globals).getProp(i)
  end repeat
end
```

Figure 20-8: Currently available global variables and their values

Now run `MyShowGlobals` in the Message window:

```
MyShowGlobals
-- "gName: "
-- "gAge: "
-- "version: 8.0"
```

VOID versus <Void>

You might have noticed that sometimes we write “the function returns `<Void>`,” and other times “set the variable to VOID.” What’s the difference? VOID is a Lingo constant used to set a variable to an uninitialized state. `<Void>` is the way a VOID value is displayed in Director. The best way to illustrate this is in the Message window:

```
put VOID
-- <Void>
```

If you have worked with the Message window, this might not convince you. Anytime an uninitialized variable is `put` to the Message window you will get a `<Void>` value:

```
put aVar
-- <Void>
```

Assuming that `aVar` was never given a value, `<Void>` is the expected result. But VOID is a Lingo constant; you can try to set a constant to a value and not get an error, but a constant’s value will never change:

```
set VOID to 1
put VOID
-- <Void>
```

The same holds true for the other constants, such as SPACE:

```
set SPACE to 3
put SPACE
-- " "
```

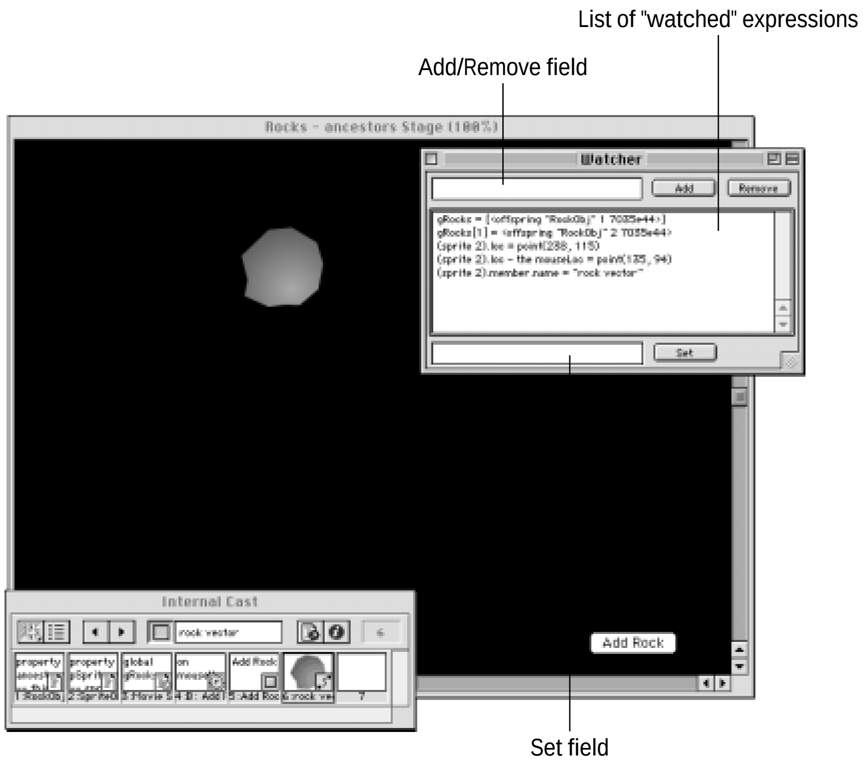
Other constants are TRUE, FALSE, EMPTY, BACKSPACE, ENTER, QUOTE, RETURN, and TAB.

Writing the constant in all capital letters is a stylistic convention; you can write it in any case you like (but you’ll find that most programmers follow the convention):

```
put VOID
-- <Void>
put void
-- <Void>
put Void
-- <Void>
```

Using the Watcher Window

The Watcher window, shown in Figure 20-9, displays the values stored in selected variables and other Lingo expressions while a movie plays. Remember that an expression is something that evaluates to a single value. This means that you can have global variables in the Watcher window, as well as expressions such as `(sprite2).loc - the mouseLoc`. To open the Watcher window, choose Window ➪ Watcher or press Command+Shift+` (Ctrl+Shift+`).

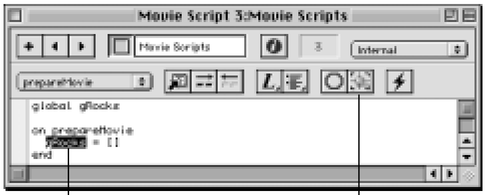


```
gRocks = [<offspring "RockObj" 1 7035e44>]
gRocks[1] = <offspring "RockObj" 2 7035e44>
(sprite 2).loc = point(238, 115)
(sprite 2).loc - the mouseLoc = point(135, 94)
(sprite 2).member.name = "rock vector"
```

Figure 20-9: The Watcher window

Designating variables to watch

Before you can use the Watcher window, you must establish the expressions that you want watched. To do this, you identify the expressions by using the Debugger window, the Message window, the Script window, or the Watcher window. In each case except the last, you use the Watch Expression button to identify the expressions to watch. Figure 20-10 identifies the Watch Expression button in the Script window. If you do not see a Script Toolbar in your Script window, choose View ➪ Script Toolbar.



```
global gFlocks

on prepareMovie
  gFlocks = []
end
```

Figure 20-10: The Watch Expression button looks the same in the Debugger and Message windows as it does here in the Script window's toolbar.

Adding variables to and removing variables from the watch list

To add a variable to the watch list using the Debugger window, the Message window, or the Script window, do the following:

1. Select the variable in the window.
2. Click the Watch Expression button or press Shift+F9.

You can tell Director to watch more than one expression at a time. As the data stored in a variable changes, the Watcher window displays the current value.

It's easy to add variables to and remove variables from the watch list in the Watcher window by doing the following:

1. Type the variable's name in the Add/Remove field (identified in Figure 20-9).
2. Click the Add button to add the variable, or click the Remove button to remove it.

Establishing values for variables on the watch list

When debugging a script, you might want to establish an initial value for a variable. This can be helpful when determining whether a specific value generates an error. To set the value of a variable, you type the value or string in the Set field of the Watcher window (identified earlier in Figure 20-9), and click the Set button.

In this next exercise, you use the Watcher window to track a variable. If you examine Score script 4 in the counter1.dir movie, you find that the `gCounter` variable increments until its value is greater than 10. By watching the value stored by the `gCounter` variable, you can see this occur.

On the CD-ROM

You can find the file counter1.dir on the companion CD-ROM in the EXERCISE:CH20 (EXERCISE\CH20) folder.

Tracking the gCounter Variable in the Watcher Window

1. Open the counter1.dir movie. You need either the movie that you modified earlier in this chapter or the file counter1.dir on the companion CD-ROM.
2. Open the Message window and click the Trace button.
3. To open the Watcher window, choose Window ➪ Watcher, or press Command+Shift+` (Ctrl+Shift+`).
4. Select the Add/Remove field (top-left corner of the Watcher window) and type **gCounter**.
5. Click the Add button. The `gCounter` variable is listed in the Variable List field located in the middle of the Watcher window. The variable's initial value might be `<void>` or zero (`0`). It doesn't matter until you play the movie.
6. In the Director toolbar, click the Rewind button, and then click the Play button to start the movie. Leave the windows open.
7. The `gCounter` value in the Watcher window first appears as follows:

   ```
   count=0
   ```

   This is based on the initial value set in the `on startMovie` handler. As the movie plays, the Watcher window displays the current value of the variable `gCounter`.
8. Click the Count to 10 button. After the script finishes running, the field on the Stage displays the value 10, and the line in the Watcher window Variable List field reads as follows:

   ```
   gCounter = 11
   ```
9. Now you reset the value of the `gCounter` variable by using the Watcher window. With the movie still playing, select the variable `gCounter` in the Variable List field.
10. Select the Set field, type **33**, and click the Set button. The new value for the variable appears in the Watcher window.
11. While the movie is playing, click the Count to 10 button. This causes the handler to execute, and the value for the variable changes again to 11.
12. Stop the movie.
13. Now let's add another variable to the watch list, but this time by using the Message window. Click the Message window to activate it.
14. Scroll up in the Message window until you locate the following expression:

   ```
   member("number").text = string(gCounter)
   ```

15. Drag across the following portion of the expression:

```
member("number").text
```

16. Click the Watch Expression button on the Message window toolbar. The Watcher window now lists a second entry that displays the text stored in the field named `number`.

17. Run the movie, and click the Count to 10 button. When the script finishes running, the Watcher window updates the values for `gCounter` and `the text of field number`.

18. Now you will remove a variable from the watch list. Select `the text of field number` entry in the Variable List field of the Watcher window.

19. Click the Remove button on the Watcher window. The Watcher window no longer displays a value for the `number` field.

20. Stop the movie.

Using the Debugger Window

You can manually open the Debugger window, shown in Figure 20-11, by choosing Window ➪ Debugger, or you can press Command+` (Ctrl+`). This window also opens automatically when the playback head encounters a *breakpoint.* (A breakpoint stops the execution of the code at that point and opens the Debugger window. Breakpoints are usually set in the Script window and are discussed shortly.)

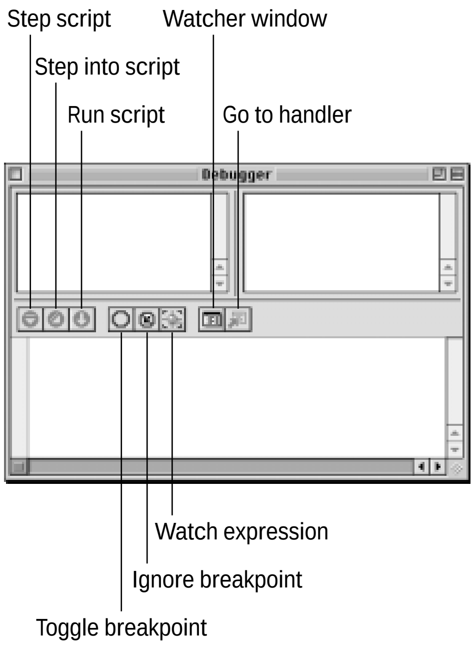


Figure 20-11: The Debugger window

From the Debugger window toolbar, you can open the Watcher window (using the Watcher window button), add an expression or variable to the Watcher window (using the Watch Expression button), and open the Script window and jump to a specific handler (using the Go to Handler button). Each of these features is also available from the Message window or the Script window, as discussed earlier in this chapter.

The Debugger window contains several controls for debugging complex scripts. In the Debugger window, you can do the following:

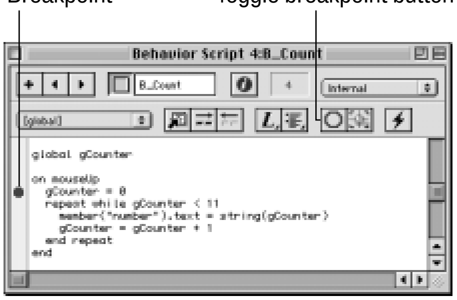
- ✥ Examine the current line of Lingo as it executes.
- ✥ Play the current handler line by line.
- ✥ Walk through the sequence of handlers as the current handler calls them.
- ✥ Debug errant code by examining the values stored in local variables, global variables, and properties as the movie executes.
- ✥ Access the Watcher and Script windows, for additional troubleshooting.

Note To actually edit or correct an error in a script, you must open the Script window. You cannot edit in the Debugger window.

Setting breakpoints within your scripts

To use the full capabilities of the Debugger window, you need to set breakpoints in your handlers. Then you can use the Debugger window to watch the breakpoints within your scripts. Setting breakpoints is a good way to determine the state of variables within scripts and handlers at specific points during the execution of a movie.

One way to set breakpoints in a Script window is to click a Lingo command and then click on the Toggle Breakpoint button, as identified in Figure 20-12. You can also add a breakpoint by choosing Control ➪ Toggle Breakpoint (or by pressing F9). When you set a breakpoint in the Script window, a red dot appears in the left column of the window (see Figure 20-12).



```
global gCounter

on mouseUp
  gCounter = 0
  repeat while gCounter < 11
    member("number").text = string(gCounter)
    gCounter = gCounter + 1
  end repeat
end
```

Figure 20-12: The Script window with a breakpoint

Tip

The fastest method of setting a breakpoint in the Script window, however, is to click in the left column adjacent to the line of code where you want the breakpoint to occur. To remove the breakpoint, click on the red breakpoint symbol.

Ignoring all breakpoints

When necessary, you can tell Director to temporarily ignore breakpoints during your troubleshooting activities. Simply click the Ignore Breakpoints button, identified in Figure 20-11, or choose Control ➪ Ignore Breakpoints (Alt+F9).

When the Ignore Breakpoints button is selected (or you choose Control ➪ Ignore Breakpoints), the breakpoint symbol still appears in the left column of the Script window, but it is grayed out. This indicates that the breakpoint is ignored in the Debugger window.

Tip

After successfully finding the problem in a script and fixing it, you should run the script again to verify that it is error-free. The easiest way to test a corrected script is to temporarily disable the breakpoints.

Removing a single breakpoint

To remove a single breakpoint, click anywhere in the line containing the breakpoint and then press F9. You can also click the Toggle Breakpoint button or choose Control ➪ Toggle Breakpoint to remove a single breakpoint. This method retains all other breakpoints whether they are in the same handler or in another handler within the movie.

Tip

The fastest way to remove a breakpoint is to click on the breakpoint symbol in the left column of the Script window. The breakpoint is erased when the red breakpoint symbol disappears.

Removing all breakpoints

After you troubleshoot and fix your scripts, you can quickly and painlessly remove all breakpoints that have been set. There's no need to open each script and toggle the breakpoints off individually. All you do is choose Control ➪ Remove All Breakpoints. All breakpoints in all scripts are removed (not just disabled, as they are when you use the Ignore Breakpoints button).

Tip

If you want to reuse a previous breakpoint after removing all breakpoints, you must reset it in the appropriate script or handler.

Running through your script in the Debugger window

The Debugger window has three additional buttons that are used to run your movie while troubleshooting each handler.

Stepping through the script

The Step Script button runs the current line of Lingo and any handlers it calls. In essence, this lets you walk through your script, executing it line by line. You can also choose Control ➪ Step Script to execute one line of code at a time, or click the Step Script button in the Debugger window's toolbar, or use the keyboard shortcut Command+Option+Shift+down arrow (press F10 for Windows users).

Running the script through the next breakpoint

The Run Script button exits debugging mode and continues execution of the handlers in your movie. In the case of the counter1.dir movie, clicking the Run Script button continues the repeat loop until the breakpoint in the repeat loop is encountered again.

The basic difference between the Run Script button and the Step Script button is that the Run Script button executes all lines of code until the line with the breakpoint is encountered again, whereas the Step Script button executes one line of code at a time. Using the Run Script button is useful when you are confident that certain parts of the script are performing accurately, and you only want to pause at specific lines of code with breakpoints. This saves you time and effort, because it eliminates the need to repeatedly click the Step Script button to move forward one line of code at a time.

Instead of using the Run Script button, you can choose Control ➪ Run Script or press F5.

Running the whole script

The Step into Script button follows Lingo's normal execution flow. It executes each line of code and follows, line by line, each nested handler that the line calls. Step into Script is handy when you're debugging errors in nested logic. For example, if you have a Lingo command that plays another Director movie, using the Step into Script button executes the current line of Lingo that calls the other movie and then jumps to that movie and executes it line by line.

The Step into Script button also works effectively for troubleshooting nested repeat loops or other control structures that use nesting.

The equivalent command for the Step into Script button is to choose Control ➪ Step Into Script or press F8.

An exercise with the Debugger window

In the following steps, you use the counterup.dir movie from the CD-ROM to set breakpoints in the Script window, and then you debug the script by using the Debugger window.

As you troubleshoot the counterup.dir movie, the problem appears to be on the line that reads as follows:

```
repeat while gCounter >=10
```

The greater-than symbol (>) rather than the less-than symbol (<) has been used. The lines that increment the `gCounter` variable are never executed because 0 (the initial value) is never greater than or equal to 10.

Repeating an action until a value is less than or equal to a specific value (such as 10) is more efficiently accomplished by stipulating that the loop continue while the count is less than one number larger than your target value. The proper method of repeating actions until the `gCounter` variable is equal to or less than 10 is to use this instruction:

```
repeat while gCounter < 11
```

While you run the practice movie, watch the position of the green arrow at the left margin. The green arrow shows the line of the Lingo script that is to be executed next. This is a helpful method of testing the logic of your control structures, such as a loop. By monitoring the state of the loop and the value of the conditional variables in the loop, you can thoroughly test the logic of the loop.

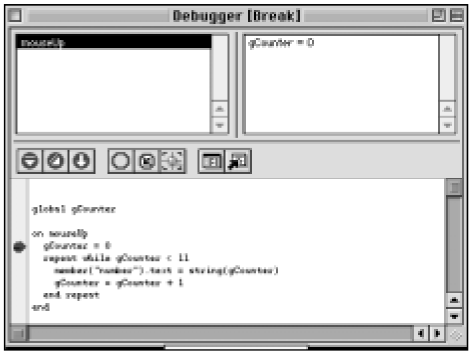
On the CD-ROM

You can find the file counterup.dir on the companion CD-ROM in the EXERCISE: CH20 (EXERCISE\CH20) folder.

Checking Handlers by Using the Debugger Window

1. Open the counterup.dir movie and play it.
2. After the movie starts playing, click the Count to 10 button. The value in the field on the Stage remains set to zero (0).
3. Stop the movie.
4. Open the script cast member stored in cast slot 4. This is the behavior attached to the Count to 10 button.
5. Click in the left column of the Script window parallel to the line of code that reads `gCounter = 0`. A red breakpoint symbol appears in the left column, identifying the location of the new breakpoint within the handler.
6. Close the Script window (and the Behavior Inspector, too, if it's open). This should catch any syntax or spelling errors, if they exist.
7. Play the movie and click the Count to 10 button. As soon as the line of code with the breakpoint is reached, the Debugger window automatically opens, as shown in Figure 20-13.

Note that the top-left pane of the Debugger window displays the name of the current handler (`mouseUp`). The top-right pane displays all variables and their values. Only the `count` variable is listed with a value set to zero. The bottom pane displays the script for the current handler. The green arrow in the left column of the bottom pane sits on top of the breakpoint symbol and identifies the point within the script where the movie has temporarily stopped. You might not be able to see the green in the screenshot of Figure 20-13.



```
gCounter = 0
```

```
global gCounter

on mouseUp
  gCounter = 0
  repeat while gCounter < 11
    member("number").text = string(gCounter)
    gCounter = gCounter + 1
  end repeat
end
```

Figure 20-13: The Debugger window opens when breakpoint is reached.

8. Click the Step Script button (or press F10) to move the Lingo script to the next command, which is `repeat while count >=10`.
9. Click the Step Script button again, and the green arrow jumps to the end keyword at the end of the `on mouseUp` handler. It skips the next two lines, which increment the value of the variable `gCounter`.
10. Click the Step Script button one more time. The Debugger window goes blank, because the playback head is no longer in the handler that contains the breakpoint.
11. Click the Count to 10 button on the Stage again. The Debugger window becomes active again, because the playback head hits the line of code that includes the breakpoint.
12. To open the Script window and make the necessary change, click the Go to Handler button on the Debugger window (the far-left button on the Debugger's toolbar).
13. In the Script window, change the handler to match the following:

```
global gCounter
on mouseUp
  gCounter = 0
  repeat while gCounter < 11
```

```
      member("number").text = string(gCounter)
      gCounter = gCounter + 1
    end repeat
  end
```

14. Leave the breakpoint on the `gCounter = 0` instruction so that you can watch the values increment in the Debugger window.

15. Close the Script window.

16. Play the movie and then click the Count to 10 button.

17. Save the movie as **counterup1.dir** by choosing File ➪ Save and Compact, or recompile all scripts in the movie by choosing Control ➪ Recompile All Scripts.

18. Play the movie and click the Count to 10 button.

19. Click the Step Script button to step through the handler, and watch the `repeat while` loop execute. To completely step through the handler will require four clicks of the Step Script button (before the `gCounter` variable increases to 1). You can watch the variable `gCounter` change values in the top-right pane of the Debugger window.

20. Continue to step through the script until the `repeat while` loop finishes and the value of `gCounter` reaches 11. At that point, if you keep clicking the Step Script button, the Debugger window will go blank because the playback head exits the `on mouseUp` handler.

21. Now you test the corrected script but cause the playback head to stop only at the breakpoint and not at each line of code. With the counterup1.dir movie still open, click the Rewind button on the main Director toolbar.

22. Click the Play button and then click the Count to 10 button. The Debugger window displays the script with the breakpoint as before.

23. This time, when the playback head pauses at the breakpoint, click the Run Script button. Instead of stopping on each line of Lingo script, the script runs until the breakpoint is encountered again. The `gCounter` variable is still updated, as you can see by examining the field on the Stage that displays the value 10 when the handler finishes.

24. Stop the movie.

On the CD-ROM

If you want to practice troubleshooting files, you can use the following three practice files: matchit.dir, orient.dir, and virgil.dir on the companion CD-ROM in the EXERCISE:CH20 (EXERCISE\CH20) folder.

Summary

You learned the following things in this chapter:

- Director has several tools to locate syntax, spelling, and logic errors in scripts.
- Syntax and spelling errors are the most common problems that you will encounter as you write Lingo scripts. You can always use the Alphabetical Lingo and Categorized Lingo buttons (in the Script window and the Message window) to assist in typing the correct syntax and spelling for a command, property, or function.
- In the Message window, you can test a single instruction or a single line of code. You can also run any handler within the current movie just by typing the handler's name into the Message window.
- To help you identify errant code, use the Watcher window to track variables and their values. You can track one variable or many by adding the variable's name to the variable list pane in the Watcher window. You can also specify a value for a variable while the movie is running, and you can watch the results as the movie plays.
- Identifying errors of logic is more difficult than locating syntax and spelling errors. You do have a powerful tool in the Debugger window — breakpoints that help you step through the execution of a handler.

The next chapter deals with memory management in Director.

✦ ✦ ✦