CHAPTER 11

Programming Fundamentals

In This Chapter

Learning programming fundamentals with Lingo

Director's four script types

Explanations of events, messages, and handlers

Many artists start working with Director and soon learn that the real power lies in Lingo. They might complete the Lingo tutorials provided with the program, but it still doesn't click for them because they haven't had any programming experience. The problem lies in understanding the very fundamentals of programming.

This chapter teaches you some programming basics. If you've never programmed before, this is the chapter for you. If you've done some programming in other languages, you might want to skim through this chapter, because it will give you a good idea of how Lingo works compared to languages with which you may be familiar. You might be surprised at some of the things that Lingo can do!

Using the Message Window

The Message window is very useful for testing small bits of code, debugging, and, in this case, teaching (see Figure 11-1). You'll want it open for the rest of this chapter, so choose Window➪ Message or press Command+M (Ctrl+M for Windows).

Figure 11-1: The Message window

In the Message window, you can type a line of code and press the Return key (the carriage return, *not* the Enter key on the keypad), and the code executes. You can do this while a

movie is running or stopped. Also, you can output information to the Message window with the `put` command.

The `put` command places strings or numbers into the Message window. When you have a great deal of information being output to the Message window, it can affect the performance of the movie. Although the Message window is available during the movie-creation process in Director, it does not appear in Shockwave or a projector. If you leave `put` commands in your code when you make a projector or Shockwave movie, it does not affect your movie's performance or cause errors.

Variables

A *variable* is a means of storing data. It is like a box that can hold one item at a time. A variable is the name of the storage location (the box). Whenever you use that name, you get the value of the data at that location (whatever is in the box). The data referenced by a variable is changeable throughout the life of the variable. As soon as you assign another value to the variable, the prior value is gone.

Type the following in the Message window and press Return at the end of the line:

```
set myNumber to 10
```

Then type this command:

```
put myNumber
```

Once again, press Return at the end of the line.

```
-- 10
```

As soon as you press Return, `-- 10` appears. Pressing the Return key in the Message window makes Director execute that line of Lingo, even when the movie is not playing. Typing `put myNumber` executes that line of code. The `put` command displays the value of the variable `myNumber` in the Message window. At that point in time, `myNumber` was holding the number 10. Now type the following:

```
set myNumber to "Hello"
put myNumber
```

Don't forget to press the Return key after each line! After you type the second line and press Return, you get the following:

```
-- "Hello"
```

You've changed the contents of the variable from the number 10 to the word "Hello." Of course, now the variable name doesn't make too much sense, but you learn about appropriate variable names a little later in this chapter.

Commenting your code

You might be wondering why two hyphens appear before the value of the variable after you `put` it to the Message window. In Lingo, a double-hyphen indicates a comment. A comment is code that is not executed. In other words, it does not do anything.

For example, type the following:

```
-- put myNumber
```

After you press Return, no value appears in the Message window as it did in the preceding exercise. Director knows that a line beginning with two hyphens is a comment and ignores it. Comments are a useful way to make your code more understandable to yourself and others.

You can also put comments on the same line as executable code. The code before the comment executes, unaffected by the comment that follows it:

```
put myNumber -- nothing after two hyphens gets executed
```

If you've been working along while reading this section, your Message window should look like Figure 11-2.

Figure 11-2: How your Message window should look

Creating variables

In some programming languages, you need to declare a variable before you can initialize it (*initialize* means to give it a value). In Lingo, a variable is created as soon as you use it. So, when you type a line of code like the following:

```
set myName to "John"
```

You've created the variable `myName`. You could have written it this way as well:

```
set myName = "John"
```

Or even like this:

```
myName = "John"
```

In the preceding two code samples, the equal sign (=) is not an equal sign but an *assignment operator*. The preceding example uses syntax first introduced in Director 7. It is more concise, but perhaps less readable, for the beginner than the older syntax, which uses the `set` keyword. In this book, we address both old and new syntax.

Naming variables

A variable name can include up to 256 characters. A good variable name, however, shouldn't be so long as to be cumbersome. It has to start with a letter (from *a*–*z*) or an underscore (_), and can be upper- or lowercase. After that, you can have numbers in the name. Variable names are only one word, so you can't have spaces in the name.

Legal variable names

Here are some variable names that do not cause a syntax error:

```
MyVariable
some_stuff
_stillValidName
aaa
```

Illegal variable names

These variable names cause errors:

```
2bad          -- can't start with a number
No Good       -- spaces aren't allowed
stillbad?     -- can't use the ? character
```

It is good to give variables meaningful names, because the name of the variable should give you an idea of the value it holds. You might be tempted to give it a meaningless name if you are in a rush or not feeling very creative. Don't do it! Although `aaa` is a legal name, it won't make sense to you a week from now. It'll be even worse for someone else trying to figure out your code.

Variable case

In some programming languages, the case of a variable is important. This is not true in Lingo. The variable name *MYNAME* is considered the same variable as *myName*.

In the Message window, type the following:

```
set myVariable to "This is my variable, I'd know it anywhere."
put MYVARIABLE
-- "This is my variable, I'd know it anywhere."
```

Types of variables

A variable in Lingo can hold data of different types: basic data, symbols, lists, and objects.

Basic types

In programming languages, different kinds of data are said to be of different *types*. Text is different from numbers, so they are considered different types of data. Numbers can be either integers or floats (decimal point numbers). Text is called a *string*. If you visualize a piece of text as a bunch of individual letters strung together, the term string should make more sense to you. Table 11-1 shows some common data types, as well as examples of each.

Table 11-1
Common Data Types in Lingo

Data Type	*Examples*
Integer	1, –34, 10000022334
Float	–0.124, 1.5, 30000.1
String	“Hello”, “a”, “Now is the time”
Symbols	#hello, #x1, #abc_def
Lists	[1, 2, 3], [“abc”, 1.2, 34], [#name: “John”, #age: 33]
Objects	<offspring “myObj” 2 53a7f7c>

Any variable in Lingo can hold any of the types available in Lingo, regardless of the type it currently holds. If one moment a variable is holding a string and a little later you decide to stick an integer in there, Lingo doesn’t complain.

Symbols

Integers, floats, and strings are obvious data types, but what is a symbol? Computers deal with numbers much more quickly than strings. Sometimes you want the speed of a number and the descriptiveness of a string. *Symbols* are Lingo’s way of satisfying this need. Symbol names have the same rules as variables, but they must begin with a pound sign (#). They are not case sensitive.

In the Message window, type the following:

```
mySymbolVar = #name
put mySymbolVar
-- #name
```

Lists

Most data types consist of one element. Sometimes, however, you might want to have a variable hold more than one item. Because variables hold only one thing at a time, you need a data type that can hold more than one item. For circumstances like these, you can use lists. We cover lists in more detail in Chapter 12, but for now you just need to know that they are a collection of elements contained within brackets, and that they come in two varieties: linear and property. The elements can be the same type or different types.

Type the following in the Message window:

```
set myLinearList to ["abc", 1.2, 34]
set myPropertyList to [#name: "Rob", #age: 40]
```

Objects

Objects can combine data and functionality. Objects hold their data in variables called *properties*. The functionality comes from handlers defined for each class of object. We cover objects in detail in Chapter 13.

Expressions

Expressions are chunks of code that result in a value. They exist within statements. An expression might be a single variable or it might be a formula. Following are examples of expressions:

```
1 + 1
power(3, 2) + power(4, 2) - 2 * 3 * 4 * cos(1)
myVar
```

They are valid expressions, but they are *not* valid statements. If you type any of these in the Message window, you get a syntax error as soon as you press Return.

Statements

A *statement* is one executable line of Lingo code. A line of code is terminated by a carriage return (the Return key). To make the preceding example expressions into statements, you can adjust the code, as shown here:

```
set mySum to 1 + 1
myAng = power(3, 2) + power(4, 2) - 2 * 3 * 4 * cos(1)
myVar = 101
```

You get no syntax errors when you type these in the Message window! These are statements because they are an executable line of Lingo (in fact, we are executing it in the Message window). `1 + 1` is an expression in the `set mySum to 1 + 1` statement.

Operators

An *operator* is a symbol or word that acts, or "operates," on one or more elements. Operators are part of the Lingo programming language and are used for calculations, assignment, and comparison.

Assignment operator

The equal sign (=) functions as a comparison operator and as an assignment operator. When you want to assign a value to a variable, you can do it four different ways:

```
set myNumber to 10
set myNumber = 10
put 10 into myNumber
myNumber = 10
```

If you've used versions of Director before 7.0, the first three should be familiar to you (unless you were avoiding Lingo). The last line, `myNumber = 10`, means the same thing. The value on the right side of the assignment operator is placed into the variable on the left side. In other words, it is *assigned* to that variable.

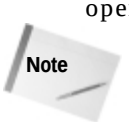
For example, type in the following:

```
myNumber = 2 + 2
```

Now `myNumber` holds `4`. `2 + 2` is evaluated to `4`, and then that value is assigned to the variable. All operations to the right of the assignment operator are carried out first.

Arithmetic operators

For doing simple arithmetic calculations in Lingo, you use the +, -, *, and / operators for addition, subtraction, multiplication, and division, respectively. If all the operands are integers, the result is an integer, including division! All arithmetic operations in Lingo that contain a float result in a float.



An *operand* is the object of an operation. If you are adding 1 + 2, the operands are 1 and 2, with the addition operator in between.

Addition

Addition uses the + operator. It requires an operand on each side of the operator. The `put` command executes on the result of the addition, putting it to the Message window.

```
put 5 + 3
-- 8
put 5.0 + 3
-- 8.0000
```

Lingo even enables you to use numbers that are strings in your calculation, as follows:

```
put "5" + 3
-- 8.0000
put "5" + "3"
-- 8.0000
```

Note that when you add two strings, the result is a float, not a string.

Subtraction

Subtraction uses the - operator. It requires an operand on each side of the operator. Again, the `put` command executes on the result of the subtraction, putting it to the Message window.

```
put 5 - 3
-- 2
put 5.0 - 3
-- 2.0000
put "5" - 3
-- 2.0000
```

Multiplication

Multiplication uses the * operator and works like the other operators:

```
put 5 * 3
-- 15
put 5.0 * 3
-- 15.0000
put "5" * 3
-- 15.0000
```

Division

Division uses the / operator. But there is more to division in Lingo than meets the eye. Look at the following example, `5/3`: The result, `--1`, is not the one that first comes to mind.

```
put 5/3
-- 1
```

Anytime all of the operands are integers, the result is an integer. In the preceding example, 3 goes into 5 only one time, so the answer is 1.

Now look at this example:

```
put 5.0/3
-- 1.6667
```

After changing at least one of the numbers to a float, you get the result you were probably expecting: `1.6667`.

You can also use strings. The result is a float even when both operands are strings:

```
put "5"/3
-- 1.6667
put "5"/"3"
-- 1.6667
```

If your expression has a variable in it, the variable is evaluated first and then the calculation is performed:

```
x = 5  -- you could also write set x to 5, set x = 5,
       -- or put 5 into x
put x - 2
-- 3

y = 2
put x - y
-- 3
```

You can have multiple operators in any arithmetic expression:

```
put 1 + 2 + 3
-- 6
put 1 + 2 - 3
-- 0
```

Modulo

The `mod` operator (short for "modulo") divides the dividend by the divisor and returns the remainder of that operation:

```
put 10 mod 3
-- 1
put 19 mod 5
-- 4
```

It's like doing integer division, but the result is the remainder. For example, 3 goes into 10 three times with 1 left over.

```
put 10/3 -- integer division
-- 3
put 10 mod 3
-- 1
```

Modulo works only with integer expressions. If an expression evaluates to a float, it is treated as an integer for the mod operator, as follows:

```
put 5.3 mod 3
-- 2
put 5.9 mod 3
-- 0
```

Lingo rounds the 5.9 automatically to 6. The remainder of 6 divided by 3 is 0.

This operator does not work with strings:

```
put "5" mod 3
-- 0
```

Mod is handy when you want to determine whether a number is even or odd. Even numbers divided by 2 have no remainder, so you know when a number is odd:

```
put 6 mod 2
-- 0
put 7 mod 2
-- 1
put 8 mod 2
-- 0
```

Negation

Most operators we've covered so far are binary operators. *Binary* operators have two operands. *Unary* operators have only one operand. The negation operator, `-`,

is unary. Putting the negation operator in front of an expression makes a positive expression negative and a negative expression positive:

```
- 2 + 3 -- the result is 1
- 3 + 2 -- the result is -1
```

Again, you can use parentheses to make it clearer:

```
(-3) + 2      -- the result is -1
((-3) * 8)/2  -- the result is -12
```

Parentheses and precedence

Arithmetic operators have an order of precedence similar to algebra. Multiplication has a higher precedence than addition:

```
put 7 + 3 * 2
-- 13
```

In the preceding example, `3 * 2` is evaluated first, and then `7` is added to produce the result of `13`.

You can use parentheses to make your code clearer, even when you do not functionally need them:

```
7 + 3 * 2
7 + (3 * 2) -- this is more easily read as 13
```

If you want the addition to be evaluated first, you can put parentheses around it, as you would do in math. Expressions in parentheses are evaluated first:

```
put (3 + 2) * 7
-- 35
```

In cases where the precedence of the operators is equal, the expression is then evaluated from right to left:

```
put 2.0 * 5 / 4
-- 2.5
```

Comparison operators

Comparison operators are also called *predicates*. Predicates are operators that return true or false. In Lingo, a value of 1 is considered true, and a value of 0 is considered false. These operators work the same way in algebra; the only one that might look unfamiliar is the Lingo version of "not equal to." Table 11-2 lists the comparison operators.

Table 11-2
Comparison Operators

Operator	*Name*
=	Equals
>	Greater than
<	less than
>=	Greater than or equal to
<=	less than or equal to
<>	not equal to

Try the following in the Message window:

```
put 5 = 5
-- 1
put 5 = 4
-- 0
```

The first statement is true, so it evaluates to 1. The second statement is false, so it evaluates to 0. Here are some additional examples:

```
put 5 > 4
-- 1
put 5 >= 5
-- 1
put 5 <> 4
-- 1
put 5 = 5.0
-- 1
```

The result of the comparison operators is a Boolean value. A *Boolean* value is either true or false (1 or 0).

These comparison operators work nicely with strings. You can use the less-than and greater-than signs to find out what is less, in terms of alphabetical order:

```
put "John" = "JOHN"
-- 1
put "John N" > "John"
-- 1
put "a" > "b"
-- 0
put "a" < "b"
-- 1
```

```
put "A" = "a"
-- 1
"A" = "a" because Lingo is NOT case sensitive.
```

Logical operators

Logical operators are operators whose operands are Boolean values, and the result is a Boolean value. A Boolean value is a value that is true or false. In Lingo, any positive or negative integer is true, and zero is false. A Boolean result is either 1 for true or 0 for false.

AND and OR

`AND` produces a true result only if both operands are true:

```
put 1 AND 1
-- 1
put (5 > 4) AND (5 = 5)
-- 1
```

`AND` produces a false result if one or both of the operands is false:

```
put 1 AND 0
-- 0
put 0 AND 0
-- 0
put (5 > 4) AND (5 = 4)
-- 0
```

`OR` produces a true result if either of the operands is true:

```
put 1 OR 0
-- 1
put (5 > 4) OR (5 = 4)
-- 1
```

`OR` produces a false result if both operands are false:

```
put 0 OR 0
-- 0
```

not

If you have a single operand and you want the opposite true value, you can use the `not` function, as follows:

```
put not 1
-- 0
```

```
put not 0
-- 1
```

Lingo also has the constants TRUE and FALSE. These constants evaluate to the integer values 1 and 0, respectively. If you prefer, you can use them in your code instead of 1 or 0, to make it more readable:

```
put TRUE
-- 1
put not TRUE
-- 0
put FALSE
-- 0
```

Lingo is not case sensitive, so typing **true** and **false** in lowercase letters works as well. As stylistic convention, however, Lingo constants such as `TRUE` and `FALSE` are in all uppercase letters.

Concatenation operators

Concatenation is putting together two strings. There are two concatenation operators in Lingo, `&&` and `&`. The first, `&&`, adds a space when you put the strings together, as shown here:

```
put "Laura" && "Lynn"
-- "Laura Lynn"
put "20" & "00"
-- "2000"
```

You can put several strings together:

```
put "May" && "28," && "20" & "00"
-- "May 28, 2000"
```

Numbers are automatically converted to strings:

```
put "$19." & 99
-- "$19.99"
put 20 & "01"
-- "2001"
```

Of course, it's okay to use variables as well:

```
set myName to "John"
put myName && "Nyquist"
-- "John Nyquist"
```

Handlers

A *handler* is a way of grouping together several Lingo statements to perform a single task. Lingo has many built-in handlers, such as `on mouseUp`, but you can also create your own custom handlers.

Suppose that you want to make the sprites in the first three channels invisible. In the Message window, type the following commands:

```
set the visible of sprite 1 to FALSE
set the visible of sprite 2 to FALSE
set the visible of sprite 3 to FALSE
```

Note: You can also use dot syntax. The following three statements achieve the same result as the preceding three statements:

```
sprite(1).visible = FALSE
sprite(2).visible = FALSE
sprite(3).visible = FALSE
```

This is a long way to go about setting three sprites to be invisible. If you had to do this many times in a program, using handlers could save you time.

Creating a Handler

1. Open a new Script window by pressing Command+0 (zero) for Macintosh users or Ctrl+0 (zero) for Windows users.
2. Click the Cast Member Properties button to open the Script tab of the Property Inspector (see Figure 11-3).

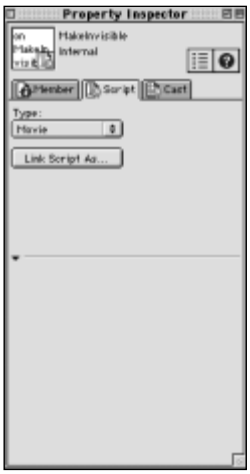


Figure 11-3: The Script Cast Member Properties tab

3. Type **MakeInvisible** into the editable field.
4. Make sure that the Type pop-up menu is Movie. If it isn't, select Movie.
5. Click OK.
6. In the Script window, type these lines:

```
on MakeInvisible
  set the visible of sprite 1 to FALSE
  set the visible of sprite 2 to FALSE
  set the visible of sprite 3 to FALSE
end
```

7. Click the Recompile All Scripts button.

A handler *definition* always starts with the word `on`, followed by the name of the handler (and any parameters), and then the statements it executes. It ends with the word `end`.

The `MakeInvisible` term did not exist in Lingo until you created it. When you type **MakeInvisible**, Lingo executes each of the statements in the handler. It does so one line at a time, from top to bottom.

Now, every time you call `MakeInvisible`, the sprites in the first three channels become invisible. Try it: Put three sprites on the Stage, and then in the Message window type:

```
MakeInvisible
```

Parameters and arguments

You can make this handler more useful by adding a parameter to it. Parameters are variables that can be used by the statements in the handler.

First, rename the handler from `MakeInvisible` to `MakeVisible`. This is an easier to understand name, given the functionality that you're going to add. Doesn't it sound more sensible to say, "It is true you want to make those sprites visible" rather than "It is false that I want those sprites invisible"? Make the following changes:

```
on MakeVisible bool
  set the visible of sprite 1 to bool
  set the visible of sprite 2 to bool
  set the visible of sprite 3 to bool
end
```

Now you've added a parameter to the handler -- `bool`. Click the Recompile All Scripts button.

In the Message window, type the following:

```
MakeVisible TRUE
```

Using the name of a handler in a line of code is referred to as *calling* the handler. When Lingo executes `MakeInvisible`, it looks up how the handler is defined. When you call the `MakeInvisible` handler, it takes the argument TRUE and puts that value into the `bool` parameter. *Arguments* are the values you pass on to parameters.

Now, if you want to make the sprites visible again, you can type the following in the Message window:

```
MakeVisible FALSE
```

Functions

A special type of handler is called a function. A *function* is a handler that returns a value.

Create a new movie script or add the following handler to the same Script window as `MakeInvisible`:

```
on CalcVolume
  set x to 10
  set y to 20
  set z to 5
  set volume to x * y * z
  put volume
end
```

Click the Recompile All Scripts button.

This handler is almost useful at this point. The problem is that it puts the result of the calculation into the Message window, where it cannot be used in further calculations. It needs to be made into a function.

Change the `put volume` statement to `return volume`:

```
on CalcVolume
  set x to 10
  set y to 20
  set z to 5
  set volume to x * y * z
  return volume
end
```

The `return` command is an important one. It breaks out of the function and returns a value to the place from where the function was called. If there were any lines of code after `return volume`, those lines would never execute.

In the Message window, type the following code:

```
put CalcVolume()
-- 1000
```

You need the `put` statement in the Message window because the `CalcVolume()` function is an expression — it evaluates to a single value. When you call a function, you need to put parentheses after it, which lets Lingo know it is getting a value back. If you forget the parentheses, the return value is `<Void>`:

```
put CalcVolume - whoops, forgot the parentheses
-- <Void>
```

This handler can be made more useful by adding parameters to it. Change the `CalcVolume` handler to the following:

```
on CalcVolume x, y, z
  set volume to x * y * z
  return volume
end
```

In the Message window, type the following:

```
put CalcVolume(10, 20, 5)
-- 1000
```

You can even perform arithmetic operations on the results of two functions, as follows:

```
put CalcVolume(10, 20, 5) + CalcVolume(5, 7, 10)
-- 1350
```

Control Structures

Control structures enable you to alter the flow of the program. Until now, the flow of the program has been from the top statement to the bottom statement without skipping any statements and doing each statement only once. Here you learn to add decision-making capability to your code with `if` and `case` structures. You also learn how to do the same thing repeatedly with `repeat` loops. There are many flavors of each, and the differences can be subtle when just reading, so make sure you work through the examples on a computer. As you become more familiar with them, the subtleties become obvious.

if structures

There is more to coding than just statements — there are decisions to be made! Statements execute one right after the other, but decision structures enable you to skip over statements, based on conditions you define.

if . . . then

The `if...then` structure enables you to make logic that executes when a certain condition exists. Here's the syntax for an `if...then` statement:

```
if integerExpression then
   statement(s)
end if
```

The words in italics mean you need to replace those words with your own code; `integerExpression` is any expression that evaluates to an integer. If that value is 0, then the statements between the `if` and `end if` do not execute. `statement(s)` is one or more valid Lingo statements.

If you try to use something that evaluates to another type, such as a float or string, you get a syntax error.

Lingo also enables you to write `if` structures with a single statement. Note that the `end if` is not needed (it *is* needed when the statement is on the line following the condition, as in the preceding example):

```
if integerExpression then statement
```

We tend to usually write it the first way (multiple lines), even for single statements. This visually separates the condition from the statements, making it easier to understand what is going on. It also makes it faster to add more statements. Also, there are times when the single-line format does not execute reliably. This bug has been documented in previous versions of Lingo. When you suspect this may be the case, try changing the `if` statement to multiple line format and see if the behavior changes.

Type the following in a movie Script window:

```
on TestTrue bool
  if bool = TRUE then
    myAns = "It's the truth"
  end if
  if bool = FALSE then
    myAns = "It's a lie"
  end if
  return myAns
end
```

If you refer to the syntax for an `if...then`, you'll notice that `integerExpression` in `TestTrue()` is the result of the equal sign acting as a comparison operator. We are testing whether the variable `bool` holds a 1 (remember, TRUE is the same as 1) or a 0 (FALSE is the same as 0). In real code, `statement(s)` is replaced by an actual statement, `myAns = "It's the truth"`.

Here is `TestTrue()` in the single-line `if` format:

```
on TestTrue bool
  if bool = TRUE then myAns = "It's the truth"
  if bool = FALSE then myAns = "It's a lie"
  return myAns
end
```

When you have multiple statements to execute based on a condition, write the `if` structure in the following format:

```
on TestTrue bool
  if bool = TRUE then
    set the visible of sprite 1 to TRUE
    myAns = "It's the truth"
  end if
  if bool = FALSE then
    myAns = "It's a lie"
  end if
  return myAns
end
```

Click the Recompile All Scripts button. In the Message window, try it out:

```
put TestTrue(TRUE)
-- "It's the truth"
put TestTrue(FALSE)
-- "It's a lie"
```

if . . . else if

When you have either-or decisions, use the `if...else if` structure, as follows:

```
if integerExpression then
  statement(s)
else if
  statement(s)
end if
```

As in the case with `if...then` structures, Lingo also enables you to write `if...else if` structures with a single statement. The `end if` is not needed (it is, however, needed when the statement is on the line following the condition, as in the preceding example):

```
if integerExpression then statement
else if statement
```

You can have as many `else if` statements as you need:

```
if integerExpression then
  statement(s)
else if
  statements(s)
else if
  statements(s)
end if
```

You can optimize the `TestTrue()` function by using the `if...else if` structure:

```
on TestTrue bool
  if bool = TRUE then
    myAns = "It's the truth"
  else if bool = FALSE then
    myAns = "It's a lie"
  end if
  return myAns
end
```

The difference here is if `bool` turns out to be TRUE, then the test for FALSE is never executed.

if . . . else

What if `bool` is neither true nor false? What if you passed in a string by mistake? Then you can add a catch-all statement at the end with one last `else` (*not* an `else if`), as in the following:

```
if integerExpression then
  statement(s)
else
  statements(s)
end if
```

If `integerExpression` is not true, then the `else` statement(s) executes. You can also put one or more `else if` statements between the `if` and the `else`. You can have many `else if` statements, but only one `else`:

```
if integerExpression then
  statement(s)
else if integerExpression then
  statements(s)
else
  statement(s)
end if
```

As in the case with `if...then` structures, Lingo also enables you to write `if...else if` structures with a single statement. The `end if` is not needed (it is needed, however, when the statement is on the line following the condition, as in the preceding example):

```
if integerExpression then statement
else statements

if integerExpression then statement
else if integerExpression then statement
else statement
```

Here is `TestTrue()` with an `else` statement added:

```
on TestTrue bool
  if bool = TRUE then
    myAns = "It's the truth"
  else if bool = FALSE then
    myAns = "It's a lie"
  else
    myAns = "I'm not really sure on this one"
  end if
  return myAns
end
```

Click the Recompile All Scripts button and try it in the Message window:

```
put TestTrue("hi")
-- "I'm not really sure on this one"
```

Nesting ifs

Sometimes you need a condition tested within a condition. In the following example, a logical `or` is used to make sure the `bool` parameter holds a TRUE or FALSE value, as opposed to some other type, such as a string, symbol, list, or object:

```
on TestTrue bool
  if (bool = TRUE) or (bool = FALSE) then
    if bool = TRUE then
      myAns = "It's the truth"
    else if bool = FALSE then
      myAns = "It's a lie"
    end if
  else
    myAns = "I'm not really sure on this one"
  end if
  return myAns
end
```

case statements

Another branching structure is the `case` statement. Like the `if...then` structures, you can write the `case` statement many different ways. Often your decision to use `case` or `if` is based on which structure you are more comfortable with.

Here is the syntax you can use that is like an `if...then` structure:

```
case (expression) of
  expression:
    statement(s)
end case

case (expression) of
  expression: statement
end case
```

Like an `if...else if` structure:

```
case (expression) of
  expression: statement
  expression: statement
end case

case (expression) of
  expression:
    statement(s)
  expression:
    statement(s)
end case
```

Or like an `if...else` structure:

```
case (expression) of
  expression:
    statement(s)
  otherwise
    statement(s)
end case

case (expression) of
  expression:
    statement(s)
  expression:
    statement(s)
  otherwise
    statement(s)
end case
```

```
case (expression) of
  expression: statement
  otherwise statement
end case

case (expression) of
  expression: statement
  expression: statement
  otherwise statement
end case
```

It is optional to put a colon after `otherwise`. It works either way. Now change the `TestTrue()` function to use a `case` statement instead of an `if else`, as shown here:

```
on TestTrue bool
  case (bool) of
    TRUE:
      myAns = "It's the truth"
    FALSE:
      myAns = "It's a lie"
    otherwise
      myAns = "I'm not really sure on this one"
  end case
  return myAns
end
```

Iteration

Iteration statements enable a program to do something repeatedly until some condition is met. Iteration is often referred to as *looping*. There are four different ways to create loops in Lingo:

- ✦ repeat while
- ✦ repeat with . . . to
- ✦ repeat with . . . down to
- ✦ repeat with . . . in

repeat while

The `repeat while` structure has this format:

```
repeat while integerExpression
  statement(s)
end repeat
```

`integerExpression` is any expression that evaluates to an integer. If that value is 0, then the statements following it do not execute. If you try to use something that evaluates to another type, like a float or string, you get a syntax error. In the code, `statement(s)` is one or more valid Lingo statements.

In a Movie script, type the following:

```
on FunWithLoops
  counter = 0
  repeat while mouseUp()
    put counter
      counter = counter + 1
  end repeat
  put "Done looping."
end
```

Click the Recompile All Scripts button, and then type the following code in the Message window:

```
FunWithLoops
```

This handler continues to execute while the mouse button is up. Click the mouse to have it stop. See the sidebar "mouseUp( ), the mouseUp, and on mouseUp" for more information on the `mouseUp()` function used in this handler.

Change the handler to this:

```
on FunWithLoops
  counter = 0
  repeat while counter < 10
    put counter
      counter = counter + 1
  end repeat
  put "Done looping."
end
```

The expression `counter < 10` uses a comparison operator. Comparison operators always have a result of 1 or 0 (true or false). The repeat loop continues executing the statements while the variable counter is less than 10; after it hits 10, the expression `counter < 10` is no longer true and the loop ends, and the rest of the handler is executed.

repeat with . . . to

Another way to repeat something a finite number of times is to use the `repeat with...to` structure, which is:

```
repeat with variable = intExpression to largerIntExpression
  statement(s)
end repeat
```

mouseUp(), the mouseUp, and on mouseUp

Did you realize that `mouseUp()` in the first `FunWithLoops` handler is a function, because of the "( )" after it? It's not a user-defined function, but a function that is part of the Lingo language. It detects whether the mouse button is currently up and returns 1 (true) or 0 (false).

You can also use the syntax `the mouseUp`, which is the way it was done in earlier versions of Director. Back then, you could find it referred to as a *system property* or a function. In the past, a Lingo expression that started with a "the" was often referring to a property of some sort and would evaluate to that property's value. Because it evaluated a value, you might think it would be called a function, but because it was evaluating to a property of a movie, sprite, and so on, it probably seemed more straightforward to refer to it as the property itself, as opposed to a function that returned that property. Whether it is the property or evaluates to the value of the property, you can use either form for the same result, as in the following:

```
put mouseUp()
-- 1
put the mouseUp
-- 1
```

Don't confuse either with the `on mouseUp` event handler, which captures the message (handles it) of the mouse button being released. When a user releases the mouse button after it has been held down, that is called an *event,* and then that message ("Hey everybody! There's been a mouseUp!") is sent through the movie. We discuss events a little later in the chapter, but you can peek ahead at Figure 11-8 if you want to see a diagram of a `mouseUp` event).

The same can be said for `mouseDown()`, `the mouseDown`, and `on mouseDown`.

Change the `FunWithLoops` handler to:

```
on FunWithLoops
  repeat with counter = 0 to 9
    put counter
  end repeat
  put "Done looping."
end
```

We can make it a more flexible handler by adding some parameters, as shown here:

```
on FunWithLoops start, finish
  -- if start is GREATER than finish, it'll skip the repeat
  repeat with counter = start to finish
    put counter
  end repeat
  put "Done looping."
end
```

repeat with . . . down to

If you need to count backward, you can use the `repeat with...down to` structure:

```
repeat with var = intExpression down to smallerIntExpression
  statement(s)
end repeat

on FunWithLoops start, finish
  -- if start is LESS than finish, it'll skip the repeat
  repeat with counter = start down to finish
    put counter
  end repeat
  put "Done looping."
end
```

repeat with . . . in

When you need to go through a list of items, you can use the `repeat with...in` structure, as follows:

```
repeat with variable in aList
  statement(s)
end repeat

on FunWithLoops
  set aList to [1, 5, "hey", 2.65, 6]
  repeat with anItem in aList
    put anItem
  end repeat
  put "Done looping."
end
```

The `repeat with...in` structure is only used with lists and objects. This structure repeats the loop as many times as there are elements in the list. It automatically puts the next element into the variable `anItem`. Of course, we can improve this handler, too, by adding a parameter:

```
on FunWithLoops aList
  repeat with anItem in aList
    put anItem
  end repeat
  put "Done looping."
end
```

Now call the revised `FunWithLoops` from the Message window and pass in a list, as shown here:

```
FunWithLoops [1,2,3]
-- 1
-- 2
```

```
-- 3
-- "Done looping."
```

If you call this version of `FunWithLoops` and pass in something other than a list, you get a syntax error.

Exiting a loop

Another way to write a repeat loop is to use the `repeat...while` structure with a literal so that it is always true. Of course, the loop would execute infinitely, unless you have a way to break out of it. That's where the `exit repeat` command comes in. The `exit repeat` command does exactly what it says — it exits the repeat. The following is an example:

```
on FunWithLoops
  -- This loop will count to 10 and then exit.
  counter = 0
  repeat while TRUE
    put counter
    if counter > 9 then
      exit repeat
    else
      counter = counter + 1
    end if
  end repeat
  put "Done looping."
end
```

Another way to modify the flow of a repeat loop is to use `next repeat`. Suppose that you only want to count even numbers:

```
on FunWithLoops
  repeat with i = 1 to 10
    if i mod 2 <> 0 then
      next repeat
    end if
    put i && "is even"
  end repeat
  put "Done looping."
end
```

When you call this in the Message window, you'll get:

```
FunWithLoops
-- "2 is even"
-- "4 is even"
-- "6 is even"
-- "8 is even"
-- "10 is even"
-- "Done looping."
```

As soon as next repeat is encountered, the loop starts back at the top. None of the statements below it are executed.

Nested repeats

You can place repeat loops within each other as well, as this example shows:

```
on FunWithLoops
  set myList to [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
  repeat with aList in myList
    repeat with anItem in aList
      put anItem
    end repeat
  end repeat
end
```

The outer loop goes through the three lists within `myList`. Each iteration places one of the lists into `aList`. The inner loop goes through the elements of `aList`.

MakeVisible revisited

The `MakeVisible` handler mentioned earlier in the chapter can be improved with the use of a repeat loop. This handler used three very similar statements; the only difference between each statement was the number of the sprite whose visibility was being changed. Because those sprites occurred in a sequential order, you can modify the handler as follows:

```
on MakeVisible bool
  repeat with i = 1 to 3
    set the visible of sprite i to bool
  end repeat
end
```

You can improve this handler further by adding parameters for the first and last sprite. By adding these parameters, you increase the flexibility of the handler dramatically; the handler is no longer limited to just sprites 1 to 3:

```
on MakeVisible bool, firstSpr, lastSpr
  repeat with i = firstSpr to lastSpr
    set the visible of sprite i to bool
  end repeat
end
```

Creating Different Types of Scripts

Director has four types of scripts. In this chapter, you've been working with the Movie script. Each script has its own effective domain where it has control. Figure 11-4 shows five scripts because frame scripts and sprite scripts are both behaviors.

Figure 11-4: The Cast window showing movie, frame, sprite, cast, and parent scripts.

Movie scripts

When handlers were introduced earlier in the chapter, it was in reference to Movie scripts. Handlers in Movie scripts are global in scope, meaning that they can be called from handlers in any of the other types of scripts and the Message window. A way to remember this is to recall that a Director document is called a movie, and that handlers in Movie scripts can be called from anywhere in the movie.

Movie scripts can include handlers that are automatically executed when the movie starts, stops, or idles. Movie scripts might also be used to define and initialize global variables for use within your movie, and to store user-defined handlers, such as `MakeInvisible`, `CalcVolume`, `TestTrue`, and `FunWithLoops`, which you created earlier in this chapter.

Behaviors (Score scripts)

A behavior is a script that is attached to sprites or frames in the Score. A behavior is sometimes referred to as a *Score script*, which refers to where behaviors are applied. Behaviors attached to sprites are called *sprite scripts*, and behaviors attached to frames via the script channel (the behavior channel) are called *frame scripts*. A sprite might have more than one behavior (script) attached to it. A frame can have only one behavior attached to it.

Whenever you attach a behavior to a sprite from the Library palette, that behavior is added to the cast. When you edit a behavior, that change takes effect in every frame and every sprite to which it is attached.

Behaviors on sprites (sprite scripts)

A sprite can have one or more behaviors attached to it. Each behavior can have one or more handlers in it. The handlers can be event handlers that are part of Lingo or they can be user-defined handlers. These handlers are executed when a message is sent to the sprite to which the script is attached. Sending a message to a sprite simply means calling a handler in one of its behaviors.

The most common use for a sprite behavior is to perform an action based on a mouse click. If the cast member (from which the sprite was created) also has a script, Director ignores the cast member script unless specific commands are in place in the sprite behavior to pass the event to the cast member script.

It's important to understand what happens when you attach a behavior to a sprite that also has a cast member script. If both scripts have an `on mouseUp` handler, the behavior intercepts the `mouseUp` event. The cast member script is not called unless the `mouseUp` in the behavior has the `pass` command in it. If there is more than one behavior attached to the sprite, and they all have a `mouseUp`, all of the `mouseUp` handlers in the behaviors execute. If you want the cast member script's `on mouseUp` to execute as well, all of the `mouseUp` handlers in the behaviors must contain the `pass` command.

As the next section ("Events and Script Priority") shows, sprite scripts can receive many events. Every time the playback head moves, every sprite script in that frame of the Score gets an `enterFrame`, `exitFrame`, `prepareFrame`, and `idle` message. Of course, the sprite does not do anything if an `on exitFrame` handler was not defined in its behavior.

To create a new sprite behavior:

1. Open the Score by choosing Window ➪ Score.
2. Select a sprite.
3. Click the Behavior pop-up menu on the Score window.
4. Choose New Behavior. A Behavior script window opens.

You've just created a behavior and applied it to the sprite at the same time.

If you close an empty, nameless script window, Director removes that member. If you want to create the member but not write anything at the moment, just give the script cast member a name or type a double-hyphen in the window. Because a double-hyphen is read as a comment, it won't be executed.

Another way to create a behavior on a sprite is to Ctrl+click the sprite (right-click for Windows users) and select Script.

A third way is to:

1. Select the sprite.
2. Open the Behavior Inspector (choose Window➪Inspector➪Behavior).
3. Choose New Behavior from the Behavior pop-up menu on the Behavior Inspector.

If you want to create a behavior without applying it to a sprite, just do the following:

1. Open a script window by choosing Window➪Script. If one is open already, click the New Script button.
2. If it doesn't say Behavior Script in the title bar, click the Cast Member Properties button and select Behavior from the Type pop-up menu.

Behaviors on frames (frame scripts)

A frame script is attached to a specific frame in your Director movie. This type of Score script (behavior) executes when the playback head reads the script channel cell in a specific frame.

Although a frame script can receive and act upon most system messages (refer to the later section "Events and Script Priority"), the most common messages handled by a frame script are `enterFrame` and `exitFrame`. Frame scripts are created and edited in the Script channel of the Score window. By default, when you create a frame script (by double-clicking an empty frame channel cell), Director inserts the beginning and ending lines for an `on exitFrame` handler.

A typical frame script reads as follows:

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

This is one of the most common and useful frame scripts. It causes the playback head to go to the current frame (loop) until some other action is invoked (such as clicking a mouse button or pressing a key).

Though the `go to the frame` statement loops through the current frame, it does not redraw the Stage unless objects change between each loop. In other words, Director only refreshes those areas of the Stage that change. Using the `go to the frame` instruction does not cause a loss in performance as other types of loops do

(`repeat with` and `repeat while` statements lock out any user interaction until they complete their work.

There are several ways to create a frame script. You can:

1. Open the Score by choosing Window ➪ Score.
2. Double-click a cell in the Script channel of the frame on which you want this behavior.

Or you can:

1. Open the Score by choosing Window ➪ Score.
2. Select a cell in the Script channel of the frame on which you want this behavior.
3. Click the Behavior pop-up menu in the Score window.
4. Choose New Behavior. A Behavior script window opens.

You also can:

1. Open the Score by choosing Window ➪ Score.
2. Ctrl+click (right-click for Windows users) in the Script channel the cell that you want the behavior to be on.
3. Choose Frame Script.

Cast member scripts

A cast member script is attached to (stored with) the cast member. A cast member can have only one cast member script attached to it. Cast member scripts are often used to hold `on mouseUp` and `on mouseDown` handlers, but they can have user-defined handlers as well.

To create a cast member script, do the following:

1. Create a button cast member from the Tool palette (choose Window ➪ Tool Palette if it is not open).
2. Highlight the cast member (choose Window ➪ Cast if it is not open).
3. Click the Cast Member Script button in the Cast window.

When you click the Cast Member Script button, the Script window opens (see Figure 11-5). Note that the title bar at the top of the Script window indicates the type of script. By default, when you create a cast member script, Director inserts the beginning and ending lines for an `on mouseUp` handler.

Figure 11-5: The Script window, showing a new cast member script

If the Cast Member Properties dialog box is already open, you can access the Script window by selecting the Script button in that dialog box.

You can see whether a cast member has a script attached to it by looking at the cast slot that holds the cast member. The presence of a script icon, as shown in Figure 11-6, indicates that a script is attached. The feature that displays the script icon can be toggled on and off. Just choose File ➪ Preferences ➪ Cast to display the Cast Window Preferences dialog box, and then place a check mark in the check box for Show Cast Member Script Icons, as shown in Figure 11-7.

Figure 11-6: The Cast Window, showing a cast member script attached to member 1

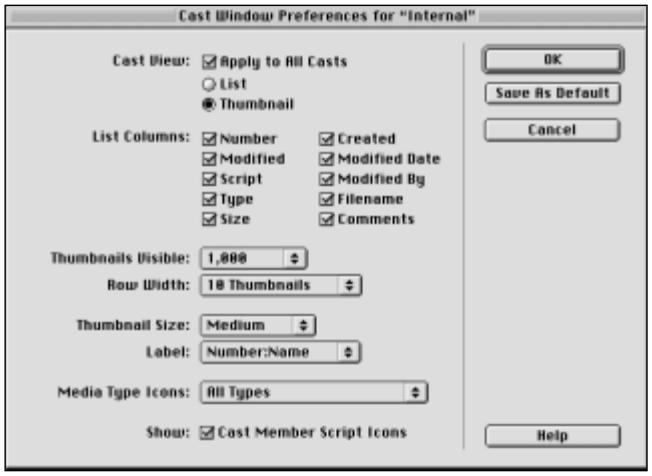


Figure 11-7: The Cast Window Preferences dialog box

Parent scripts

Parent scripts are special scripts used by Lingo to create child objects. When a script has been specified as a parent script, its handlers have the scope of the Script window, like a behavior. A parent script differs from a behavior in that the script does not appear in the Behavior Pop-up menu of the behavior or in Sprite Inspectors. (Parent scripts are used in object-oriented programming and discussed at length in Chapter 13.)

Behaviors, movie scripts, and parent scripts appear in the Cast window as cast members, but cast member scripts do not; they appear as an attachment to a cast member. Scripts that appear as cast members are accessible from the Script pop-up list in the Score window or by double-clicking the script cast member in the Cast window.

Events and Script Priority

You might have noticed that all four script types can receive and act on some of the same messages (such as the `mouseDown` and `mouseUp` messages), so you might be wondering what happens when a handler exists for the same event. What if a movie script has a handler that is meant to intercept a mouse click, as does the current frame script, cast member script, and sprite script? Which script gets control first?

An event is some action that occurs in a program. An event can be the action generated by a user typing or clicking the mouse, or it can be generated within Director when the playback head moves, or it can even be generated by a handler. Table 11-3 lists the events defined in Lingo and the scripts that are sent messages when such an event occurs.

Table 11-3
Messages, Related Events, and Script Types

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on activate Window`	A window is selected or becomes active.	X				
`on alertHook`	This event enables you to do error handling.					X *

Continued

Table 11-3 *(continued)*

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on beginSprite`	The playback head enters a frame containing the first cell of a sprite. (A *cell* is the intersection of a channel and frame. A sprite might span more than one cell). An `on beginSprite` initializes a sprite's properties. It occurs before it's drawn and it occurs only once (if the playback head leaves and returns, it is called again).			X	X	
`on closeWindow`	A window is closed.	X				
`on cuePassed`	A cue point in a sprite or sound is passed.	X	X	X	X	
`on deactivate Window`	A window is deselected or becomes inactive.	X				
`on endSprite`	The playback head leaves the last frame of a sprite's span. An `on endSprite` handler cleans up after the sprite restore properties, and to compact memory.			X	X	
`on enterFrame`	The playback head enters the frame prior to the frame being drawn.	X	X	X	X	

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on exitFrame`	The frame has been drawn, but the playback head hasn't yet moved to the next frame. An `on exitFrame` handler frequently contains navigational information that tells where the playback head should go next.	X	X	X	X	
`on EvalScript`	In a Shockwave movie, this message is received from a browser.	X				
`on getBehavior Description`	This event is called when the Behavior Inspector is opened. It returns a string used as a description of the script.	X	X			
`on getProperty DescriptionsList`	This event is called when you drag a behavior to the Score or you double-click the behavior in the Behavior Inspector. The result is used for the parameter's dialog box.	X	X			
`on hyperlink Clicked`	This event is called when a user clicks a hypertext link.		X	X		
`on keyDown`	The user presses a keyboard key.**	X	X	X	X	
`on keyUp`	The user releases a keyboard key.**					

Continued

Table 11-3 *(continued)*

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on idle`	The movie has no other instructions.	X	X	X	X	
`on mouseDown`	The primary (left) mouse button is pressed. This message can be intercepted by the sprite or cast member over which the mouse pointer rests, or it can be intercepted by the frame if you just want to capture a mouse click.	X	X	X	X	
`on mouseEnter`	The mouse moves into the intercept region of the sprite (usually the sprite's bounding box unless the sprite uses the Matte ink). This event only occurs once, when the mouse first enters the sprite's intercept region and until a `mouseLeave` event occurs.			X	X	
`on mouseLeave`	The mouse pointer exits the intercept region of a sprite, regardless of whether the mouse button is pressed.			X	X	
`on mouseUp`	The mouse button is released. This event is sent each time the primary (left) mouse button is released.	X	X	X	X	

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on mouseUp Outside`	The mouse button is released outside a sprite's bounding box, after the mouse button was originally pressed while over the sprite.			X	X	
`on mouseWithin`	This event occurs repeatedly while the mouse pointer resides within the bounding box of a sprite. Use this event handler only when absolutely necessary, because the constant stream of `mouseWithin` messages intercepted by a handler can slow movie performance.			X	X	
`on moveWindow`	This event is called when you drag a movie's window.	X				
`on openWindow`	A window opens.	X				
`on prepareFrame`	This event is called when the playback head moves; it occurs before anything is drawn on the screen.	X	X	X	X	
`on prepareMovie`	This is the first message sent to a movie. This is a good place to initialize global variables.	X				
`on resizeWindow`	The user resizes a window.	X				

Continued

Table 11-3 *(continued)*

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on rightMouseDown`	The user presses the right mouse button. For Mac users, this message is sent if the user is pressing the Control key while the mouse is clicked (if `emulateMultiButtonMouse` is set to TRUE.	X	X	X	X	
`on rightMouseUp`	The user releases the right mouse button. For Mac users, this message is sent if the user is pressing the Control key while the mouse is clicked (if `emulateMultiButtonMouse` is set to TRUE).	X	X	X	X	
`on runPropertyDialog box`	This event is called when the Parameters dialog box in the Behavior Inspector is opened. It enables you to override the defaults from the `getPropertyDescriptionList`.	X	X			
`on startMovie`	A movie starts. Useful for initializing sprites in the first frame.	X				
`on stepFrame`	When the Stage gets updated, this message is sent to the `actorList`. Any child object in the `actorList` receives this message.					X

Message	*Event That Causes the Message*	*Movie*	*Frame*	*Sprite*	*Cast*	*Parent*
`on stopMovie`	A movie ends. Useful for cleaning up external resources, memory, and storing information about the session.	X				
`on streamStatus`	This handler is called when `tellStream Status` is set to TRUE.	X				
`on timeOut`	Nothing is happening. This event measures a period of inactivity and is used to trigger another event.	X				

* An `alertHook` message is sent on a global (movie-wide) level, but the handler must be placed in a parent script.

** If the `keyUp` or `keyDown` event occurs over an editable field, all types of scripts can intercept the message. If the event does not occur over an editable field, the message can be intercepted only by a frame or movie script.

Messages generated by events in a Director movie follow a specific hierarchical path. Don't worry about duplicate handlers in overlapping scripts; after a message is intercepted by one handler, it won't be passed to another handler unless specifically instructed to do so. The path of any event message is illustrated in Figure 11-8. In this example, the event is a mouse click, the message is `mouseUp`, and Director is seeking an `on mouseUp` handler to manage the event.

You can establish primary event handlers for four common events that generate messages in a Director movie (`keyDown`, `mouseDown`, `mouseUp`, or `timeOut`). If a primary event handler has been defined, the message is intercepted and the specific instructions in the primary event handler are executed first, before any of the handlers in the appropriate script in the message hierarchy. While all other event handlers automatically trap the event by default, the primary event handler passes it on by default, so you do not need to use the `pass` command.

Here's what happens when a mouse click occurs (refer to Figure 11-8). This step-by-step explanation shows how messages are generated by events and passed to handlers.

- **Primary event handler:** When the mouse button is clicked on the Stage, Director first checks to see if a primary event handler exists. If so, Director executes the instructions in the handler.

  The message is then passed, unless there is a `stopEvent` command in the handler.

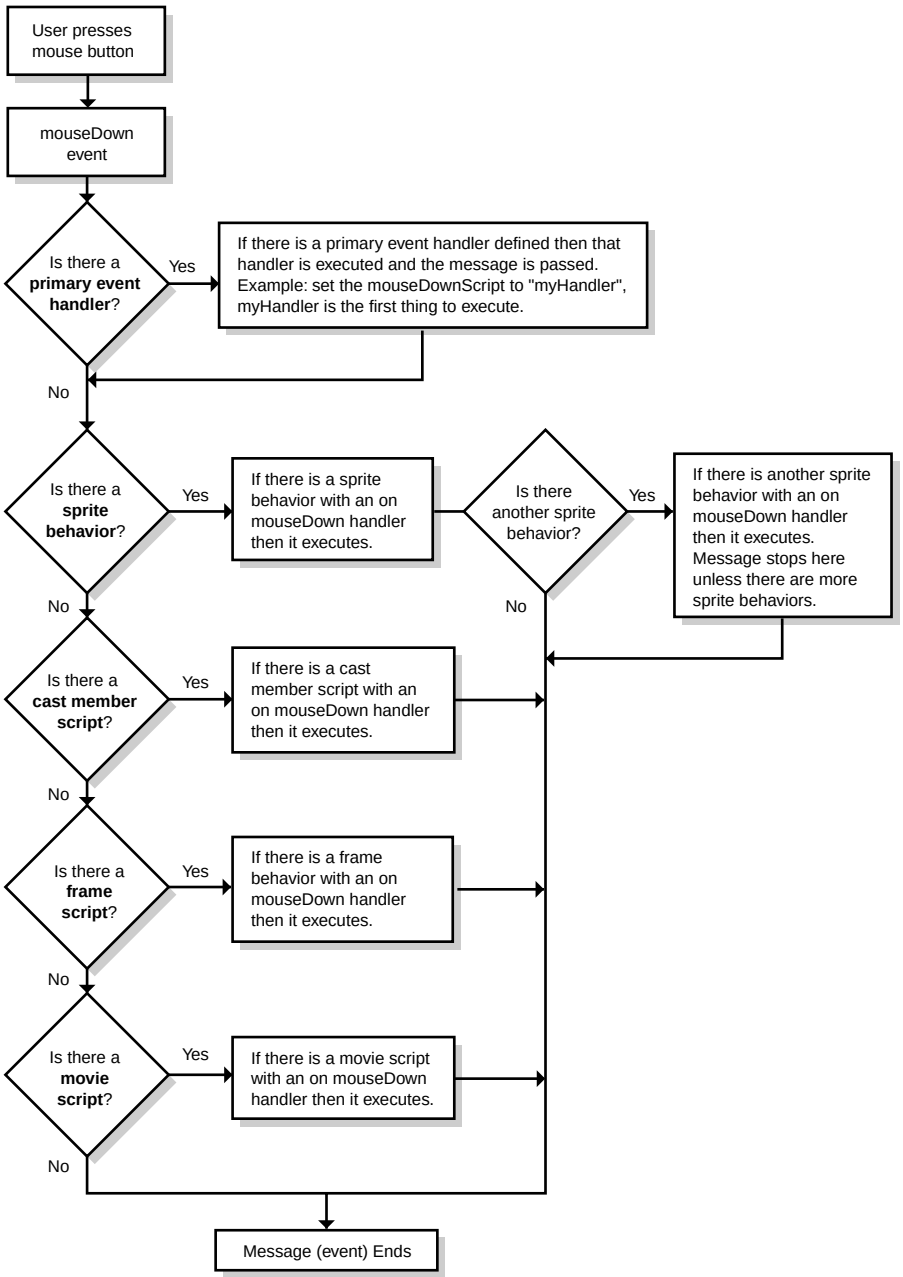


Figure 11-8: The path of a mouseDown event

If no sprite exists at the location of the mouse click, then Director immediately looks to the frame script channel to see if the frame includes a handler for the mouse click event. If no frame script exists, Director looks for a movie script that contains a handler for the event.

- **Sprite behavior:** Next, Director checks to see if there's a sprite at the clicked location. If so, Director looks for a behavior attached to the sprite. If the sprite has multiple behaviors attached, all of them are sent the `mouseDown` message. Director takes no further action, unless each of the behaviors attached has a `pass` command in the handler.

  If no behavior script exists, or if it doesn't have a handler for the event, Director checks for a cast member script.

- **Cast member script:** If a cast member script is found and it has a handler for the event, Director takes no further action, unless the handler has a `pass` command.

  If there is no cast member script, or if it doesn't have a handler for the event, Director checks for a frame script.

- **Frame script:** If a frame script is found and it includes a handler for the event, control is passed to that handler. Director takes no further action, unless instructed to by the frame script.

  If no frame script exists with a handler for the message, Director checks for a movie script.

- **Movie script:** If a movie script is found and it includes a handler for the event, control is passed to that handler.

Finally, if no script is found with a handler for the event throughout the hierarchy, it is simply ignored.

Visualize the script types as a series of safety nets, with the sprite behavior being the first net, the cast member script being second, the frame script being third, and the movie script being the final net. If a script of a certain type exists, the net captures the event (a primary event handler has no net). If the script does not exist with a handler for the event, no net exists. The event jumps off a platform and keeps falling until it encounters a safety net. When a safety net is encountered, the event is grabbed by a handler that takes its message and executes a specific action.

Following are a few additional guidelines to help summarize the priority given to various scripts.

- If you click a sprite that has both a cast member script and a sprite behavior attached, the sprite behavior takes priority.
- If you click a sprite that has a cast member script, and the current frame also has a script attached, the cast member script takes priority.

- If you click a sprite that has both a sprite behavior and a cast member script, and the current frame also has a script attached, the sprite behavior takes priority.
- If you click an area of the Stage that does not have a sprite, the frame script takes priority.
- If you click an area of the Stage that does not have a sprite and no frame script exists for the current frame, the movie script takes priority.

In the preceding statements, we used a mouse click as the event. The event could just as easily have been leaving a frame (`exitFrame`), entering a frame (`enterFrame`), no action (`idle`), the elapsing of a specified period of time (`timeOut`), or any of the other events listed in Table 11-3.

Variable Scope

Variables come in three varieties. A *global variable* can be used anywhere within a movie. A *local variable* is used only within a specific handler. After the handler completes execution, the local variable ceases to exist. Whether local or global, the first occurrence of a variable in your script creates it. The third variety is called a *property* variable. It is declared at the beginning of a behavior or parent script and is available within the handlers within that script window.

Global variables

The following guidelines are helpful when using a global variable:

- You can declare a global variable in any handler or in any script in a movie.
- If you want to establish a global variable, use the `global` keyword followed by the name of the variable. You can declare more than one global variable on the same line. The following are examples of two global statements that declare, respectively, a single variable and multiple (three) variables:

```
global gMyNewVariable
global gCost, gRetail, gProfit
```

- To use a global variable in a specific script, you must repeat the global statement within that script. Once declared, the current value or string stored by a global variable is available for use and modification within the handler.
- Sometimes, to easily differentiate between local and global variables, Lingo programmers begin global variables with the lowercase letter g.
- To see a display of global variables and their current values, use the `showGlobals` command. The result appears in the Message window.
- You can reference all of the initialized global variables with `the globals` property. This `the globals` property returns a property list of all of the global variables in use.

Local variables

The following guidelines are helpful when using a local variable:

- If you do not declare a variable as global, it is a local variable.
- A local variable can be used only within the handler in which it is declared. When the handler finishes, the variable ceases to exist.

For the sake of comparison, a local variable is like a three-year-old child, and a global variable is like a teenager. The effective domain of a three-year-old (local variable) is limited to inside the house and a fenced backyard (the current handler). The effective domain of a teenager (global variable) is anywhere and everywhere he or she decides to go (the entire movie). Both global variables *and* teenagers perform assigned tasks only when called by name (sometimes loudly), whenever and wherever they are needed.

Properties

You can use properties in both behaviors and parent scripts. The value of a property variable persists as long as the object that contains it exists. Properties declared at the top of a Script window, outside of any handler, can be accessed from any handler within that Script window. We discuss them in detail in Chapter 13.

The life cycle of a local variable

Variables have a certain (albeit very limited) kind of life. Within a script, a variable is born, lives out its life, and dies.

Birth: You write a `set` or `put` statement, with the variable being assigned a value. For example:

```
set count=0
```

In this case, Director creates space for the variable and reserves the variable name for as long as the variable shall live.

Life: The variable's contents can be accessed or changed. Within a routine, the variable retains its contents until those contents are explicitly changed. For example:

```
set count = count + 1
```

The contents of the variable `count` are updated every time the statement is evaluated (executed).

Death: Director removes the variable name from its references, the contents of the variable are discarded, and the variable ceases to exist. Death usually occurs only when the routine where the variable is defined ends. When this happens, the variable is said to "go out of scope."

Summary

Among the things you learned in Chapter 11 are the following:

- ✦ The Message window is your tool for testing code and checking properties.
- ✦ Variables can be declared (by their use) as local to a specific handler or global (available in all handlers within a movie).
- ✦ Like other programming languages, Lingo uses commands, expressions, functions, variables, and constants as code-building blocks.
- ✦ Handlers enable you to execute many lines of code with one word.
- ✦ Functions are handlers that return a value.
- ✦ Operators are symbols that tell you how two or more expressions should be combined. Lingo includes arithmetic, comparison, logical, and a few miscellaneous operators.
- ✦ There are four types of scripts in Director: movie, behavior (score/frame/sprite scripts), parent, and cast member.
- ✦ There are three types of variable scope: global, property, and local.

In the next chapter, you learn about lists and their usefulness.

✦ ✦ ✦