CHAPTER 13

Object-Oriented Programming with Lingo

In This Chapter

Lingo as an object-oriented programming tool

Relationships: Parent scripts, child objects, and ancestors

Building child objects in memory

Creating multiple child objects from a single parent script

Using the actorList

If you're new to programming, it probably won't help to compare and contrast object-oriented programming (OOP) with structured programming. OOP is a method of solving a logical problem. It's a way of thinking about the problem at a higher level—closer to how humans think. While OOP sometimes takes a little extra coding, you might find it makes programming significantly easier.

Suppose that you want to create a space game. In it, there's a spaceship, big rocks, small rocks, and laser pulses. The spaceship has weapons, engines, and shields. The weapons might be variable: You might be able to have more than one or only one. Those things, in turn, have actions that they can take. A ship can fly, shoot, be hit, and explode. Note how you are thinking about the game. You're not thinking in terms of sprites, the Score, handlers, and functions. You are thinking about things—objects.

What Is Object-Oriented Programming?

The idea of OOP has been around since the late 1960s, so it is not a new idea, but it has gained widespread popularity in the last ten years or so. With OOP, you create structures in memory that model real-world objects. The object is the basis of OOP. To be considered object-oriented (OO), a language should support abstraction, encapsulation, inheritance, and polymorphism.

Abstraction

Abstraction is another term for getting away from the details. A computer, for example, is an abstraction of a monitor, box (CPU), and modem. A monitor is an abstraction of glass, wires, plastic, and metal. When you think of a computer, you don't think of it as chips, wires, metal, and plastic; instead, you group all of that together and think of it as the item called a computer. The computer has functionality that comes along with that thinking. You can create files and directories, you can run programs, and you can communicate via the modem.

Object-oriented programming enables you to think at a higher level (more abstract), which makes problem solving simpler. In the space game mentioned earlier, you think about the spaceship moving through space, avoiding rocks, and you think about rocks being hit and broken by laser pulses. It is much easier to design and discuss the desired functionality of your program by thinking like this than by talking about a sprite of a bitmap cast member changing its loc every exit frame while watching for collisions. You still have to handle that sort of detail at some point, but that's after you know how things in the program should interact.

Encapsulation

Encapsulation is hiding information. The idea of encapsulation is that your object contains data that only it can access through its own handlers. Some languages enable the programmer to enforce encapsulation by enabling the programmer to define some of the data as touchable only by the object itself. Lingo does not support this technique; any encapsulation you do with Lingo is by design.

Encapsulation of data has benefits in protecting an object from having its data incorrectly changed from outside sources. By not exposing the data, you also prevent errors to outside sources reading incorrect data. Good encapsulation can make bug tracking easier, because you have a better idea of where to look for the problems. It also makes changing the program easier. If you want to change what your spaceship's weapon does, you just look in the script for the weapon and change it. You don't need to search through a bunch of disparate handlers, looking for references to your weapon — you know exactly where to find them.

Inheritance

Inheritance is the capability of an object to inherit data and actions from another object. With inheritance, you can define objects that are very general and have common functions such that classes that are more specific could inherit the common functionality and have additional functionality. Lingo supports inheritance through an object's ancestor property.

Polymorphism

Polymorphism refers to having different objects with the same interface, but the implementations are different. Suppose that you had car and plane objects, and each of these understands the `move` command. When you tell the car to move, it drives. When you tell the plane to move, it flies. Each knows how to move, but each moves in a different way. You don't need to know the details of how they move; you just need to know *that* they can move.

Parent Scripts and Child Objects

Objects, such as lists and rects, are not something you see on the Stage. They are structures in memory. In Lingo, objects made from parent scripts are called *child objects. Parent scripts* are like templates: They tell you what properties and handlers an object has. When you create an *instance* (child object) of a parent script, you give that instance values for the properties.

Comparisons with C++

Lingo implements objects in more than one way. First, Lingo uses parent scripts to create child objects. If you're coming from another object-oriented programming language, such as C++ or Java, this approach might seem the most familiar to you. Check out Table 13-1 for some similarities between C++ and Lingo.

In OOP, objects combine data with actions. In Lingo terms, variables are the data and handlers perform the actions. Let's start with the most basic object you can create (see also the sidebar "new( ) versus rawNew( )" below).

Table 13-1
Lingo OOP and C++ Comparisons

Lingo	*C++*	*Notes*
Parent script	Class	The parent script is different from other movie scripts because the cast member must be named (to be called by the `new` function) and contain a `new` handler with instructions for creating an object. It also can contain property declarations.
Child object	Class instance	A child object can have properties, and it can be controlled and modified by using Lingo scripts. Child objects exist only in memory, not in the Score or cast.

Continued

Table 13-1 *(continued)*

Lingo	*C++*	*Notes*
Property variable	Instance variable	A property variable is the parent script's equivalent of a global variable. Property variables persist after the parent script handler executes—just like global variables. Further, each child object can maintain its own values for each property variable. In other words, each child object (though created from the same parent script) can assign different data to the same property variables.
Handler	Method	A handler in a parent script is no different from a handler in a movie script. The structure, functions, keywords, commands, and instructions available are common to all Lingo handlers. Most of the time, the first parameter of a parent script handler is the variable `me`.
`Me`	this	`me` is a reference to the child object.
`Ancestor`	Superclass	`ancestor` is an object property used to enable child objects to inherit handlers and properties from other objects. It contains a reference to another child object.

Creating an Object

1. Open a new script window by pressing Command+0 (zero) or Ctrl+0 (zero).
2. Click the Cast Member Properties button to open the Property Inspector.
3. Select Parent from the Script Type pop-up menu under the Script tab in the Property Inspector.
4. Name the member **basic** in the Name field.
5. Click OK.
6. Type in the script window exactly what is in Figure 13-1.

Figure 13-1: The world's most basic parent script

The basic parent script has only one handler, the `new` function. We know `new` is a function because of the return statement: It returns a value. The `new` function has one parameter: `me`. This is a Lingo convention, calling the first parameter of a parent script handler `me`. It can really be any name, but it will be less confusing if we stick to the convention. The `me` parameter is a reference to this object. In the Message window, type the following:

```
myObj = new(script "basic")
```

When you call the new function of the script basic, `(script "basic")` is the argument to that function. Director creates a unique instance of this script in memory. Now type this:

```
put myObj
-- <offspring "basic" 2 2838d70>
```

Note the result when you used the put command on `myObj`. It puts the reference to the object, which tells you it is an offspring (a child) of the parent script basic. The last item, `2838d70`, is a unique identifier for this object. It will be different on your machine, and different every time you create a new object.

What are objects?

Objects are very similar to property lists. A good way to understand what an object is and why you'd want to bother with one is to start with a property list. The address book program in Chapter 12 uses property lists. Each list has the format `[#name: "", #street: "", #city: "", #state: "", #zip:""]`. In the address book program, we created a list of these lists, but for now, just create a single variable holding one of these person lists, as follows:

```
JohnDoeList = [#Name: "John Doe", \
               #Street: "123 Main St. ", \
               #City: "Anytown", \
               #State: "CO", #Zip: "12345"]
```

To put this list into five different text cast members, it was necessary to call a handler in a separate movie script. That handler was written to deal with that kind of script. It won't work with a list that has different property names. It is tailored for the address book record list. Another problem is that the handler is separated from the data on which it acts. Those two pieces of the program are somewhat separate, yet they are tightly interdependent. You can also create more handlers to deal with this kind of list. Those, too, would go in movie scripts and might be interspersed with other movie script handlers that have nothing to do with this kind of list. In other words, this situation calls for an object.

new() versus rawNew()

Director 8 introduces the `rawNew()` function. Using `rawNew()`, you can create an instance of a parent script without invoking its `new()` handler. This method is about 50 percent faster than creating an instance using the most basic `new()` handler. So, technically, the most basic parent script is an empty one!

A script can be instantiated by using `new()` even if it does not have a `new()` handler. If you compare the speed of creating instances by using `new()` without the `new()` handler in the script versus `rawNew()`, you'll still find `rawNew()` to be about 10 percent faster.

Why would you want to bypass the `new()` handler? Suppose that you want to create hundreds of instances of objects, but do not need to initialize their properties until later, on an individual basis. Using `rawNew()` would be a much more efficient way to do it. Then you can call the `new()` handler when you want to or you can directly initialize properties.

Creating a Parent Script for Person Objects

1. Open a new script window by pressing Command+0 (zero) or Ctrl+0 (zero).
2. Click the Cast Member Properties button to open the Property Inspector.
3. Select Parent from the Script Type pop-up menu in the Property Inspector.
4. Name the member **person** in the Name field.
5. Click OK.
6. Type in the script window exactly what is in Figure 13-2.

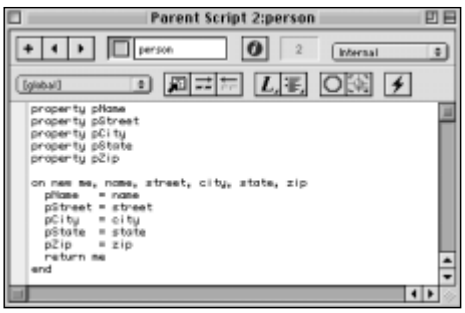


Figure 13-2: The person parent script

The person script has five properties, just like our property list person. Each property is a variable that needs to be declared with the keyword `property`.

With global variables, we use the prefix *g;* with property variables, we use the prefix *p.* This is just a stylistic convention—they can be any legal variable name. The reason we do this is that the property variable type has the scope of this entire script window; any handler inside this script window can access these properties. Because the use of the property may be in a handler much further down in a script window, it is nice to know, at a glance, that a particular variable is a property without having to scroll all the way back to the top of the window. Another benefit is that you don't have to worry about accidentally using a Lingo keyword.

The scope of a property variable is greater than that of local variables (whose life span is the length of the handler they are in). Yet, the scope of a property variable is less than that of a global, which can be accessed from anywhere in the movie—even in subsequent movies. The properties are on their own lines in Figure 13-2, but they could all be on the same line as the word *property,* separated with commas, as shown in Figure 13-3.

Figure 13-3: Another way to declare properties in a parent script window

We usually do it the first way (as in Figure 13-2), because it enables us to put comments right after the variable name, so it could read as follows:

```
property pName   -- first and last name
property pStreet -- complete street address
```

Right now, there is only one handler, which is the `new()` function. It takes five parameters, whose values we promptly put into property variables before returning from the function.

We want to be able to put a child object created from this parent script into the five text fields, as we were doing with property lists. Therefore, we will write a handler, and the handler will go into the same script window as the new function and properties. Add the `PutInfo` handler, as seen in Figure 13-4.

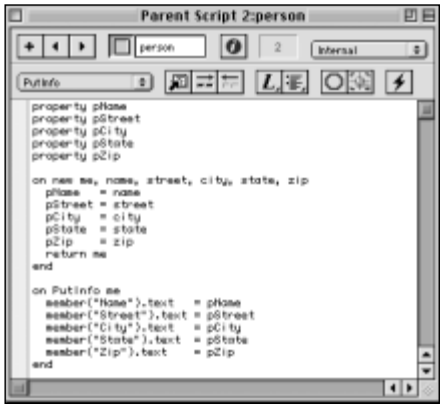


Figure 13-4: A PutInfo handler is added to the person parent script.

The nice thing about this is that we are grouping the action (the handler) with the data (the object's properties). This object now knows how to take its data and put it into the five text cast members. No one else is messing with this object's data. That is the encapsulation aspect of OOP. The object's data is encapsulated, and the only one to `get` or `set` this data is the object's own handlers. You are allowed to access an object's properties outside of the object (breaking the concept of encapsulation) with Lingo, but that is based on how you write your code.

Now create a `JohnDoeObj` object as follows:

```
JohnDoeObj =  new(script "person", "John Doe", \
                  "123 Main St", "Anytown", "CO", "12345")
put JohnDoeObj
-- <offspring "person" 2 2838eec>
```

You can see from "putting" the object in the Message window that it is the child object of the person script. This person object knows how to do one thing: put its information into the five text cast members. Now, tell it to do just that, as follows:

```
JohnDoeObj.PutInfo()
```

If you don't want to use dot notation, write the following instead:

```
PutInfo JohnDoeObj
```

As mentioned in Chapter 12 on lists, we like to use parentheses, so we would write the preceding statement as shown here:

```
PutInfo(JohnDoeObj)
```

Parentheses are not required with the syntax, `Handler(Object)` because it is not a function, but we like them there to improve readability. Visually, parentheses group the handler with the object better. Another visual clue is that handlers are defined with an uppercase letter. The parentheses are required for the dot notation syntax `Object.Handler()`. If you left them off, Lingo would think you are trying to access a property of the object instead of calling one of its handlers.

So, what have we done here? We've taken the usefulness of a property list as a way to store data, and we've coupled it with functionality. There's more to it than just that, however. We've created a *template* of a person record of an address book. It's like creating your own special kind of list, one that you can add to as well as change how it functions. Yes, an object in Lingo is like a property list in many ways and can use property list functions. The list operators `[]` work on objects. The dot notation, property list functions, and the `the propertyName of object` syntax all work for objects as well, as shown here:

```
put JohnDoeObj[#pName]
-- "John Doe"
put JohnDoeObj.pStreet
-- "123 Main St"
put JohnDoeObj.getProp(#pCity)
-- "Anytown"
put the pState of JohnDoeObj
-- "CO"
```

We hope these similarities help you feel more comfortable with working with objects.

Let's create another child object of the person parent script:

```
JaneSmithObj = new(script "person", "Jane Smith", "987 Central
Ave.", "Othertown", "AZ", "98765")
put JaneSmithObj
-- <offspring "person" 2 28390f4>
JaneSmithObj.PutInfo()
```

We now have two objects made from the same parent script. Each object has its own space in memory. They are instances (child objects, or offspring) of the same template (parent script). They have the same properties as each other, but the value each of those properties contains is different. For one object, the `pName` property holds "John Doe"; for the other, it holds "Jane Smith."

Besides these familiar functions and operators, we can now create new handlers that this object can respond to, as we did with `PutInfo`. Now add the `Introduce` handler to the end of the person script (see Figure 13-5).

Figure 13-5: The Introduce handler is placed at the end of the script person.

Caution

All variables created in the Message window are global variables. Globals persist between different Director movies. Global variables stay in memory until the program ends, the command `clearglobals` is used, or `VOID` is assigned to the variable. Objects created before this change was made, ones that are still in memory, such as globals, do not recognize the `Introduce` handler. When you create a child object, the object contains a copy of the script. Because our `JaneSmithObj` was created in the Message window before we added the `Introduce` handler, the object does not recognize the handler. It has an older copy of the person script. You have to re-instantiate the object to enable the functionality.

Re-instantiate `JaneSmithObj` as follows:

```
JaneSmithObj = new(script "person", "Jane Smith", "987 Central
Ave.", "Othertown", "AZ", "98765")
```

This is a new object in the `JaneSmithObj` variable. Now "she" understands the new handler, `Introduce`, as shown here:

```
Introduce(JaneSmithObj)
-- "My name is Jane Smith, I am from Othertown."
```

Revisiting the address book program

With the person parent script and a few changes to some movie script handlers, you can change the address book program from Chapter 12. Copy the person script cast member, open your address book program, and then paste it into the cast. Then make the changes shown in Figure 13-6 to the handlers shown.

Figure 13-6: With minor changes to four movie scripts, the address book is now object oriented.

The biggest change is in the `GetStarterList` handler. There, you create and return (in the same statement) a list of three new person objects. In `ChangeRecord`, you've replaced a cryptic repeat loop with the statement: `aPerson.PutInfo()`. Doesn't it seem easier just to tell the current person object to `put` its information? In the statement just above that one, you used the list operators instead of the `getAt()` function to access the current object. You didn't need to change the other two handlers, `AppendRecord` and `SetRecord`. Functionally, they would still work, but we thought it would read more clearly if we changed the name of the local variable `newRecord` to `newPerson`.

Controlling sprites with objects

Chapters 14 and 15 go further into sprite manipulation through Lingo. For the moment, we'll give you a brief introduction. The parent script SpriteObj, shown in Figure 13-7, has three handlers: the obligatory `new()` function, plus the `SetUpSprite` handler and the `GetChannel()` function.

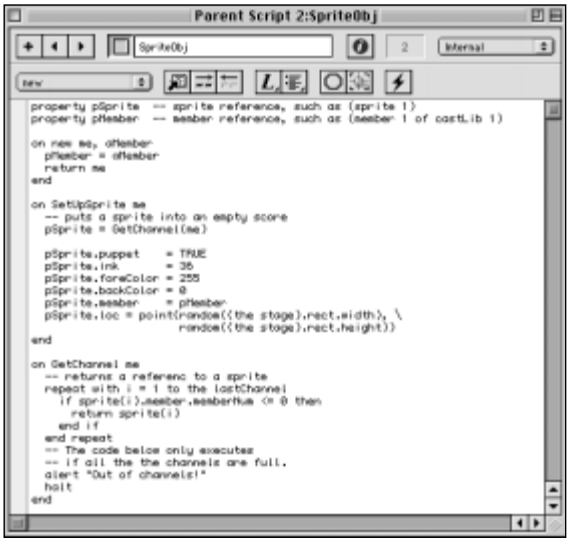


Figure 13-7: The parent script SpriteObj

Creating an Object that Controls a Sprite

1. Create a new parent script, as shown in Figure 13-7.
2. Create a cast member to use as a sprite (vector, bitmap, text, and so on) and name it **myMember**.
3. Create a `go to the frame` script in the frame script for frame 1.
4. Play the movie. It is now looping on frame 1.
5. In the Message window, type the following:

```
mySprite = new(script "SpriteObj", member "myMember")
mySprite.SetUpSprite()
```

6. Your member appears on the Stage in a random position.

Many experienced Director users are still unfamiliar with the technique of putting sprites into an empty Score. This technique isn't new: we've been using it since version 4.0 of Director. The trick is to puppet the sprite and set the sprite properties shown in `SetUpSprite`. In earlier versions, you needed to set the type of sprite as well.

Creating ancestors

Objects have the capability of inheriting behaviors and properties from other objects. Lingo has an object property called `ancestor`. Unlike the variable `me`, this is not just a naming convention. When an object receives a message (one of its handlers is called), it looks to see whether it has that handler; if it doesn't, you get a syntax error. If that object has the `ancestor` property, however, the message will be sent to the `ancestor`. Again, if that object doesn't have the handler, you get an error. Of course, the `ancestor` can have `ancestors`, and the message will be sent all the way up.

Inheritance is one way to reuse code. A good example is the SpriteObj parent script created in Figure 13-7. It is generic and lends itself to being an ancestor object. This would be a good place to start building the space game.

Creating an Object with an Ancestor

1. Create a parent script named RockObj, as shown in Figure 13-8. The important thing to look at is the initialization of the ancestor property in the `new()` function. After it creates an instance of a SpriteObj parent script, it calls `SetUpSprite` with `me` as an argument. Because it uses `me`, Lingo expects to find `SetUpSprite` in this object's script. This handler does not exist in the RockObj script, but it does have an ancestor as a property, so Director knows to look in the ancestor script next to see whether it has the handler. Sure enough, it does. The `prepareFrame` handler rotates the rock and moves the sprite.

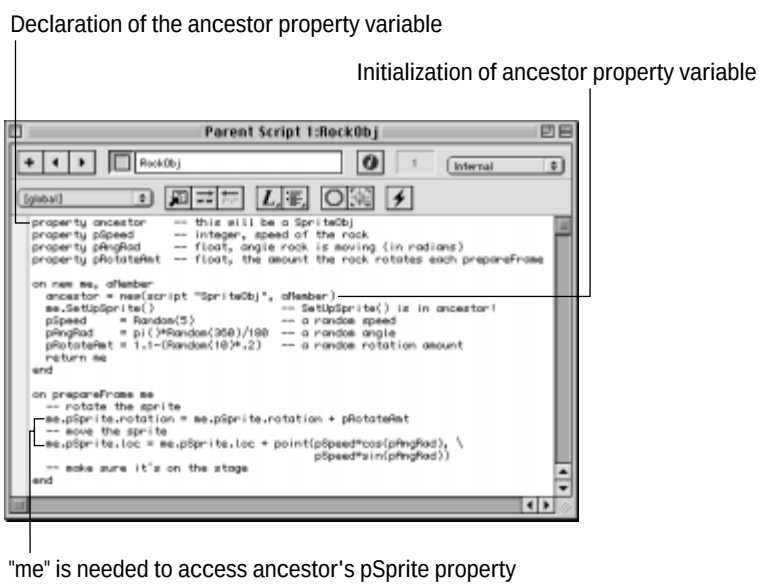


Figure 13-8: The RockObj parent script. Don't worry if you're foggy on trigonometry—just type what you see.

2. Create movie scripts as shown in Figure 13-9. The `prepareFrame` event handler initializes the global variable `gRocks` to an empty list. The `AddRock` handler takes a reference to a member as a parameter. It creates a new instance of a RockObj and adds it to the list of rocks. The `prepareFrame` event handler loops through all of the rock objects in `gRocks` and sends each a `prepareFrame` message. Then it loops on the frame. A `prepareFrame` handler can be in a movie script. If a frame doesn't have a `prepareFrame` handler of its own to trap the `prepareFrame` event (as the playback head moves), the event is passed to the movie script.

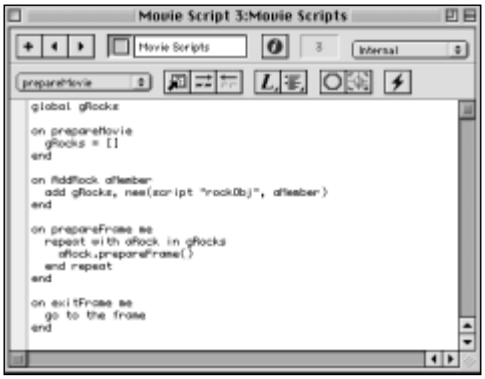


Figure 13-9: The movie script window with three simple handlers

3. Create a rock made from a vector shape and name it **rock vector**.
4. Add a button to the Stage and name it **Add Rock**.
5. Add a behavior to that button by pressing Ctrl+click➪Script (right-click➪Script).
6. Type the behavior as shown in Figure 13-10. The only statement in the `mouseUp` event handler is a call to the `AddRock` handler. It passes in one argument, a reference to a member.

Figure 13-10: The behavior for the button sprite

7. Play the movie.

8. While the movie plays, click the Add Rock button (see Figure 13-11).

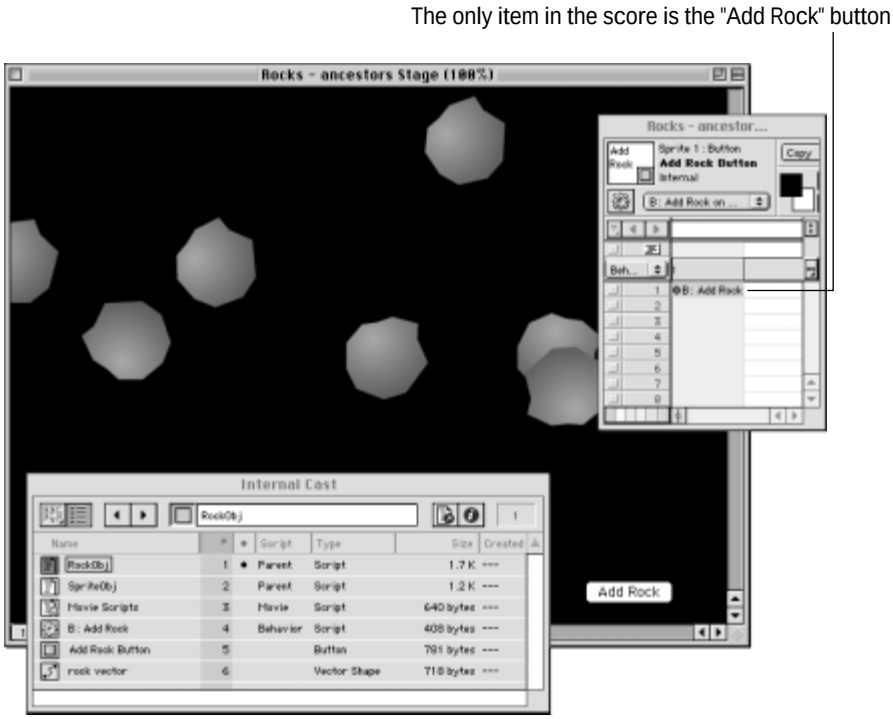


Figure 13-11: The movie playing with a few rocks added. Note that the only thing in the Score is the button. The exitFrame handler is in the movie script.

This is a neat little program, but there is one problem with these rocks: They fly off the screen, never to be seen again. Make an additional handler, called `CheckBounds`, and call it from the `prepareFrame` handler, as shown in Figure 13-12.

To some extra-logical readers, it might seem odd to name a parent script something with *Obj* as a suffix, because the parent script itself is not an object but a template from which objects are made. When used in a line of code, however, it reads quite well. For example, where we say `ancestor = new(script "SpriteObj", aMember)`, what we want in ancestor is an object of the SpriteObj script.

9. Save this movie as **Rocks – ancestors.dir.**

Figure 13-12: Add the handler CheckBounds to make sure that the rocks wrap around to the other side of the screen.

Using the actorList

The `actorList` is a movie property that is global in scope. It is used in conjunction with child objects. Child objects added to the `actorList` automatically receive a `stepFrame` event every time the Stage is updated. Objects remain in this list even when you go to other movies. It can be cleared by setting it to an empty linear list. You can quickly change the Rocks movie to use the `actorList`.

To use the `actorList` in the Rock Movie:

1. Create a duplicate of the file Rocks – ancestors.dir movie you just saved.
2. Open the RockObj parent script and rename the `PrepareFrame` handler to **StepFrame**, as shown in Figure 13-13.
3. Remove the global variable `gRocks` from the movie script window. Change the `prepareMovie` handler so that it sets `the actorList` to an empty list. Change `AddRock` so that it adds the new rock to `the actorList` . Delete the repeat loop from the `prepareFrame` handler (see Figure 13-14).
4. Play the movie and add some rocks. The result is the same as earlier.

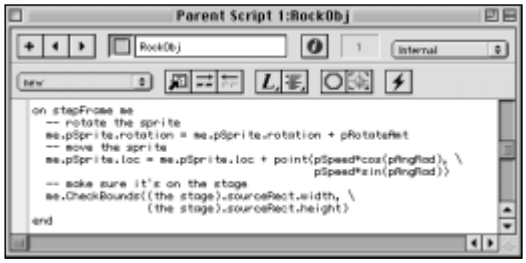


Figure 13-13: The RockObj parent script's PrepareFrame handler is renamed StepFrame.

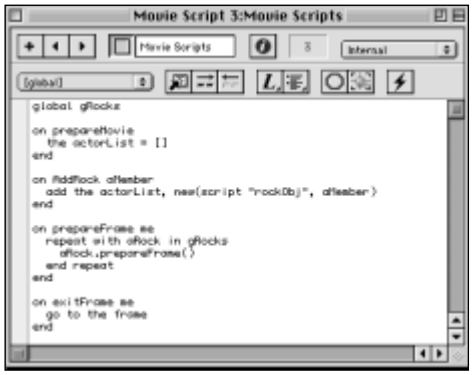


Figure 13-14: All three movie scripts have minor changes. Note that the global variable gRocks is no longer needed.

Advantages of the actorList

Well, you rewrote the Rocks movie so that it does the same thing. Which way is better?

- Using `the actorList` gives you a way to automatically send `stepFrame` events to an object.
- A `stepFrame` is generated when the Stage is updated by the playback head moving or when the `updateStage` command is used.
- Using `the actorList` avoids your having to declare a global variable.
- The scope of `the actorList` already is global.
- We've heard claims of `the actorList` being a faster way of sending messages. We don't have hard data on the validity of these claims and we haven't seen humanly noticeable results when we've compared the two methods. Still, it's worth trying if you are trying to squeeze more speed out of a movie.

Disadvantages of the actorList

Yes, there is a downside to `the actorList`. Most of the time, we choose to use the first method for these reasons:

- The global nature of `the actorList` enables the objects in it to persist even when you go to a new movie. They will continue to get `stepFrame` events, possibly causing havoc. Child objects in a global list that go to another movie will do nothing unless their handlers are called.
- By using a global list to selectively send messages, such as `prepareFrame`, you can easily enable and disable the activity of your objects throughout your movie.
- `the actorList` can turn into a dumping ground for objects of different types.
- Clearing `the actorList` may be unwise at the start of a movie, because you might end up getting rid of an object from a previous movie that was supposed to be there. It is almost as bad as calling `clearglobals`.

Using parent scripts instead of globals

As mentioned in this and earlier chapters, a global variable can persist between movies. Sometimes this is not the behavior you desire. When you create a movie in a window (MIAW), you might not want to use global variables, because the Stage or other MIAWs might inadvertently use them. One way around this problem is to create a parent script with properties that you want to use like global variables. You then can use the following syntax:

```
set the propName of script "myScript" to aValue
```

`propName` is the name of the property you want to set; `myScript` is the name of the parent script; and `aValue` is the value to which you are setting the property.

Advantages of using parent scripts in place of globals

This method of storing data has several advantages over using global variables:

- You can access the value from anywhere in the movie without having to declare a global.
- Your values will not show up when a `showglobals` command is issued.
- Your values exist only in the current movie.
- Your values are hidden from people examining your movie better than with globals.

Disadvantages of using parent scripts in place of globals

Some of the advantages of this technique can be disadvantages as well:

- Your values are limited to the current movie.
- You need to write specific code to `put` these values to the message window, instead of just being able to issue a `showglobals` command.
- Recompiling clears out these values.
- The code to get or set the value is longer.

This technique is similar to using class variables in other languages. If you would like to take this strategy further, see the sidebar "Class Variables in Lingo."

Class variables in Lingo

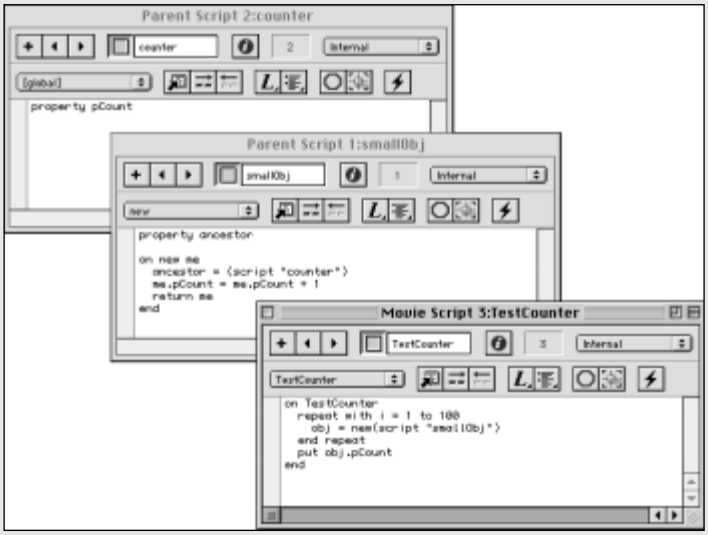
If you've programmed in C++, you are probably familiar with class member variables and methods, as opposed to instance variables. Class variables are static: the value one holds is shared across all instances of the class. You can do something similar in Lingo. Instead of setting an instance of a class as a variable, set it to a referenceto a script, as in the following:

```
ancestor = (script "sharedInfo")
```

This technique enables you to have properties share the same value across child objects of the same parent script. Why would you want to do this? Suppose that you are instantiating hundreds of objects and you want to save memory where possible. Instead of having each object hold the same piece of data (and that piece having to be written to hundreds of objects when you want to change it), you have only one that is shared by all. You could use an object that is a global variable as the ancestor, and have all of the objects share the same instance, but using a global would break encapsulation. If you're an OOP diehard, breaking encapsulation is probably distasteful to you. Besides, who wants all those globals floating around between movies and filling the Message window when you do the `showglobals` command? Here's a simple use for a static variable (see the following figure)

Continued

Continued



TestCounter creates 100 instances of the parent script smallObj.

When you try TestCounter in the Message window, you get this result:

```
TestCounter
-- 100
```

Each time through the loop, a new instance of the script `smallObj` is created. Each time a new instance of `smallObj` is created, it increments the `pCount` property of its ancestor script, `"counter"`. Because the ancestor script is a reference to a script instead of an instance, all of the objects created shared the property `pCount`. Each object, therefore, "knew" how many instances of `smallObj` were created, from the first to the last. For example, if we change `TestCounter` to that shown in the figure below, and execute it from the Message window, you get this result:

```
TestCounter
-- 3
-- 3
-- 3
```

One of the nice aspects of this method is that the values are not cleared after recompiling as they are if done something like the following:

```
set the pCount of script "counter" to 10
put the pCount of script "counter"
-- 10
```

Recompile the scripts by choosing Control➪Recompile All Scripts:

```
put the pCount of script "counter"
-- <Void>
```

This variation on TestCounter shows the creation of three independent objects, all of which reference the same script counter variable via inheritance.

Behaviors and Child Objects

Behaviors have an important relationship to OOP and to parent scripts and child objects (see Chapters 14 and 15 for a more detailed discussion). In Director, parent scripts and behaviors are almost identical in their implementations. Objects can be written with or without sprites in mind. Behaviors are written with the intention of being used with sprites.

Converting a parent script to a behavior

Parent scripts can be adapted easily to work as behaviors, and the reverse, converting behaviors to parent scripts, can also be done. Let's do that with our Rock movie. Open a copy of your Rock movie. Make the changes to the RockObj parent script as shown in Figure 13-15.

1. Remove the ancestor declaration `property ancestor` from the top of the screen, and add `property pSprite` in its place.
2. Remove the first two statements in the `new()` function and add `pSprite = sprite(me.spriteNum)` in their place.
3. Click the Cast Member Properties button to open the Property Inspector.
4. Click the Script Type pop-up menu under the Script tab in the Property Inspector, and select Behavior.

5. Technically, we're finished with this script. Yet, make one more change. Every occurrence of `me.pSprite` in the `prepareFrame` and `CheckBounds` handlers can be changed to just `pSprite`. Although `me.pSprite` will still function, you are making the movie do extra work. Because the property `pSprite` is now in the same script window, it is unnecessary to use the `me` reference. In fact, accessing via the `me` reference is a little slower.

Figure 13-15: The RockObj parent script has become a behavior.

Now you can delete the other scripts. You no longer need the SpriteObj parent script or the movie scripts. All you need now is a `go to the frame` on the frame script. We could have kept the one in the movie script, but because all of our other work is in the Score, it makes sense to have this script there as well (see Figure 13-16).

Figure 13-16: Place a simple go to the frame script in script channel at frame 1.

You can also delete the Add button, because we are not dynamically adding sprites with behaviors (although it is possible). We will use the Score to add a few rocks, and then apply the `RockObj` behavior. If you don't see the script available in the Behaviors pop-up menu, then you might have forgotten to change the type of script to Behavior. Parent scripts (as well as movie and cast member scripts) do not appear in the Behaviors pop-up. Now, just play the movie, as shown in Figure 13-17.

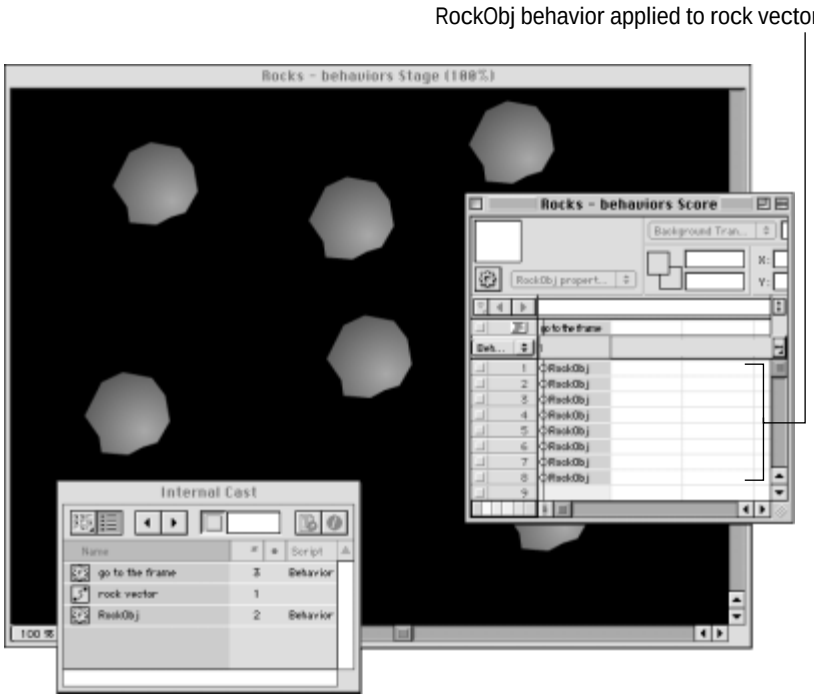


Figure 13-17: Although the behaviors version of the Rocks movie has only three cast members, there are sprites in the Score.

Because that was so easy, let's add a spaceship!

Adding a Spaceship to the Rocks Movie

1. Create ship artwork with a vector shape.

2. Put the ship artwork into the first available sprite channel.

3. Create the behavior shown in Figure 13-18.

Note that the ship uses the same statement as the rock to rotate it, but it bases the direction to rotate on user input. Since Director 7, the `keyPressed()` function has had new functionality. It enables you to detect whether a key is pressed based on the keyCode (as shown) or on a string of a single character, and returns 1 or 0 (TRUE or FALSE). This enables you to detect whether a key is down, even when other keys are pressed as well. In the `ExitFrame` handler, we'll rotate the ship if the left or right arrow is pressed, but we'll also accelerate the ship if the up arrow is pressed. This was impossible to do in previous versions of Director.

Figure 13-18: The ShipObj behavior illustrates the use of keyPressed() functionality for user input.

4. Apply the behavior to the ship sprite.
5. Play the movie. Not bad for a movie with only five cast members (see Figure 13-19)!

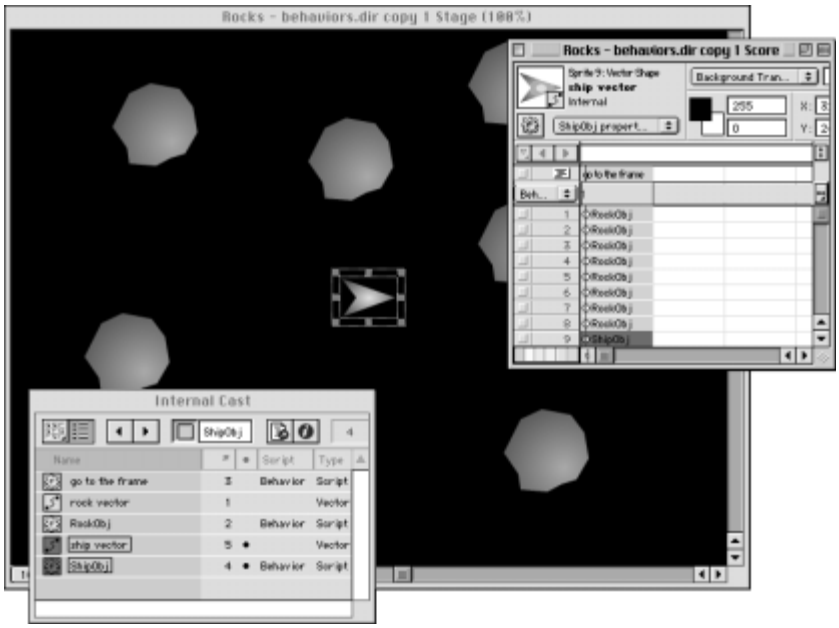


Figure 13-19: The Rocks movie now has a new inhabitant.

You can make one further optimization. You are able to specify the number of sprite channels in your movie (see Figure 13-20). You can have as many as 1000, but you can improve performance by only using what you need. For this movie, we specified 12.

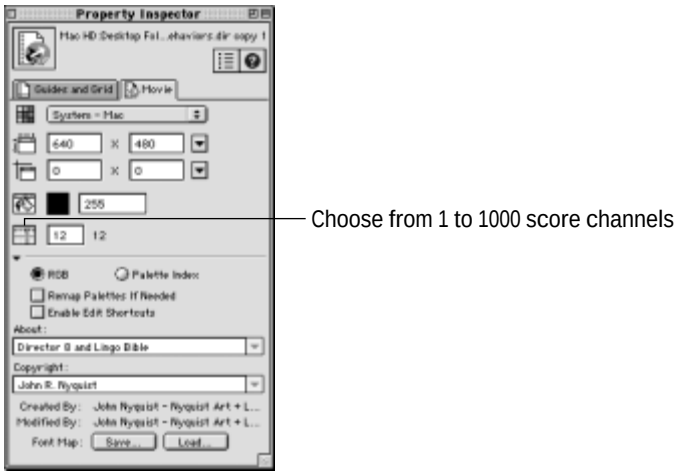


Figure 13-20: You can specify the number of sprite channels in your movie.

This example illustrates the convenience of behaviors. They enable you to think in object-oriented ways, but they greatly simplify the implementation. While we are not doing anything as advanced as dynamically putting sprites in the Score or using inheritance, we have achieved our goal in an elegant, easy-to-understand manner. This doesn't mean that you should throw out parent scripts; they have their uses in controlling sprites, cast members, and nonvisual data. After you've got your functionality set with behaviors, you can always easily modify them to make them parent scripts. In fact, behaviors are a good way to prototype your parent scripts.

Other offspring tricks

Another interesting trick that you can do with the children of parent scripts and behaviors is to use them interchangeably in the `scriptInstanceList` of a sprite. All sprites keep a list of the behaviors attached to them, which are kept in the `scriptInstanceList` of the sprite. At any time, should you so desire, you can add other instances (objects) to the list. They do not have to be instances of behaviors; they can be instances of parent scripts. By the same token, you can get any script instance of a sprite and put it into a variable and call its functions and access its properties from the variable.

Cast Member Scripts

Cast member scripts have, for the most part, been supplanted by behaviors. In fact, you can't even get to the cast member script anymore by Ctrl+clicking (right-clicking). You can still reach them, however, through the Cast Member Script button on the Cast window and by clicking the Cast Member Script button on the Member tab in the Property Inspector.

Using cast member scripts does have similarities to object-oriented programming. You are combining data (a cast member) with actions (the cast member script). You are able to create your own handlers inside a cast as well as use many of the event handlers that sprite behaviors support (the exceptions are `beginSprite`, `endSprite`, and `new`).

In versions 4 through 6.5 of Director, cast member scripts supported property variables. As of version 7, however, the properties no longer remember their values, making this feature, for the moment, dead.

Summary

Object-oriented programming enables the programmer to think about the problem at a higher level. In this chapter, you learned that:

- ✦ Lingo is an object-oriented language.
- ✦ Parent scripts and child objects have similar counterparts in C++ and Java.
- ✦ Child objects are similar to property lists, and that you can use property list functions on them.
- ✦ You can define your own handlers in parent scripts.
- ✦ In Director, the ancestor property implements inheritance and facilitates code reuse.
- ✦ Parent script properties can be used like class variables in other languages.
- ✦ Behaviors and parent scripts have a great many similarities, and it is easy to convert between the two.
- ✦ Behaviors are a rapid way to create objects.

Chapter 14 discusses the subject of building buttons.

✦ ✦ ✦