CHAPTER 12

Lists and Other Structures

In This Chapter

Working with linear lists

Working with property lists

Working with points and rects

Working with colors and dates

Sometimes the data you manage in a movie doesn't fit into a neat, single value or string, but instead must be stored and manipulated as a collection of items. Lingo responds to this need with lists. From collections of numbers to collections of windows and objects, the *list* is a tool for working with aggregate information. Being familiar with lists is very important to becoming a good Lingo programmer. This chapter teaches you to build and work with linear and property lists—as well as other Lingo structures, such as rects, points, colors, and dates—and how they are useful in your movies.

Using Lists

Lists are complex data structures that can store more than one piece of data per variable and treat the entire collection as a single item. You store and retrieve data in a list based on the data's position within the container.

Lingo supports two types of lists:

- **Linear lists:** Values in a linear list are accessed by their position in the list.
- **Property lists:** Values in a property list are accessed by a property associated with the value.

Each of these types of lists can be either sorted or unsorted. After you use the `sort` command on a list, the list remains sorted even when you make changes to it.

Director 8 has improved the access times of nonsorted lists to about equal that of sorted lists for most cases. In fact, overall list access times have improved.

Lists are valuable when you need to create an undetermined number of variables while the movie runs. Unlike variables that can hold only a single value or string at one time, lists are, by definition, variables that can store multiple values or strings at the same time.

You can add an item to a list, retrieve an item from a given position in the list, delete an item from a list, and even step through the list one item at a time. The list is actually one of the more useful structures in programming. It provides a way to put items in order, to transport a lot of items at once, and to treat all the items within the list in a similar fashion without establishing a separate variable for each item individually.

A list can hold any data type that a simple variable can — an integer, a string, a reference to a cast member, even another list.

Lingo offers several commands to manage and manipulate data within lists. They enable you to:

- ✦ Add data at the beginning of or at a specific location in a list
- ✦ Append data at the end of a list
- ✦ Remove data from a list
- ✦ Access data from a specific location in a list
- ✦ Edit data stored in a list
- ✦ Sort data in a list
- ✦ Count the number of data items in a list

Remember these rules about list commands:

- ✦ In all cases, the `myList` argument refers to the list you are modifying.
- ✦ If the data you add is a string, be sure to enclose it within double quotation marks. If the data is numeric, quotation marks are not used.

Table 12-1 identifies the most important list-related Lingo. It's been separated into categories: Lingo that works on property *and* linear lists, and Lingo exclusive to linear *or* property lists.

Table 12-1
List-Related Lingo

All Terms	*Type of List Used In*
`Add`	Linear
`AddAt`	Linear
`AddProp`	Property
`Append`	Linear
`Count`	Both
`DeleteAt`	Both
`DeleteOne`	Both
`DeleteProp`	Property
`Duplicate()`	Both
`FindPos()`	Both
`FindPosNear()`	Both
`GetAProp()`	Property
`GetAt()`	Both
`GetLast()`	Both
`GetOne()`	Both
`GetPos()`	Both
`GetProp()`	Property
`GetPropAt()`	Property
`Ilk()`	Both
`List()`	Linear
`ListP()`	Both
`SetAProp`	Property
`SetAt`	Both
`SetProp`	Property
`Sort`	Both
`Max`	Both
`Min`	Both
`[ ]` (brackets)	Both
`.` (dot)	Both

Working with linear lists

Like other variables, a list must have a name and must be initialized either with or without data. You declare, or initialize, a list when you use it. You can use two forms to declare a linear list. The more common syntax is to use the `set` command:

```
set myList to [item1, item2, item3...]
```

myList is replaced with the list's name; and item1, item2, and so on, are replaced with the data that you want stored in the list. The square brackets signal Director that the enclosed data is a list.

The other method of declaring a linear list, which is seldom used, is:

```
set myList to list(item1, item2, item3...)
```

This second method uses the `list` function, to which the data in the parentheses are returned as a list.

In the following example, you initialize (create) a linear list with three entries:

```
set products to ["motherboard", "chip set", "keyboard"]
```

Just like the different ways you learned to initialize a variable in Chapter 11, you can initialize a list by using the variety of syntactical methods Lingo supports:

```
set products = ["motherboard", "chip set", "keyboard"]
put ["motherboard", "chip set", "keyboard"] into products
products = ["motherboard", "chip set", "keyboard"]
```

The same is true for the `list()` function:

```
set products to list("motherboard", "chip set", "keyboard")
set products = list("motherboard", "chip set", "keyboard")
put list("motherboard", "chip set", "keyboard") into products
products = list("motherboard", "chip set", "keyboard")
```

Using brackets is the most common form of list creation, so from this point on, that is what we'll use in our examples.

In addition to strings, lists can include values, as in

```
set playerScores to [50, 27, 33, 66]
```

Lists do not have to be homogeneous — a single list can hold many different data types.

```
set person to ["John Smith", 34, #sales]
```

Most of the time, you find yourself using linear lists to hold elements of the same data type. Property lists lend themselves to holding different types.

To initialize a linear list that is empty, you can do any of the following:

```
set products to []
put [] into products
products = []
```

Additionally, you can use the `list()` function, which returns an empty list when passed with no arguments:

```
products = list()
```

Getting values

There are many ways to get information from a linear list. Some of them require functions, others require just the list operators `[ ]`.

All of the list functions in Lingo work on the assumption that the first element in the list is item number 1, the second is item number 2, and so forth. If you've learned programming in the C language arena, this can throw you, because C-like languages (including C++ and Java) usually start counting with zero instead of 1.

getAt() and [ ]

Both `getAt()` and the list operators enable you to get a value of a list, based on the items index in the list.

Syntax:

```
getAt(myList, index)
myList.getAt(index)
myList[index]
```

`myList` is the name of your list or a variable holding a list. `index` is an integer or a variable holding an integer. The list operators, `[]`, go at the end of the list when you want to retrieve an element from the list. The index indicates which element in the list is retrieved. Here's an example you can try in the Message window:

```
daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
put getAt(daysOfWeek, 3)
-- "Tue"
```

Or, using the dot operator:

```
put daysOfWeek.getAt(4)
-- "Wed"
```

Or, with the brackets:

```
put daysOfWeek[5]
-- "Thu"
```

This next instruction sets the variable `thirdDay` equal to the third item in the `daysOfWeek` list:

```
thirdDay = getAt(daysOfWeek,3)
```

If you type **put thirdDay** in the Message window, the result is as shown here:

```
put thirdDay
-- "Tue"
```

Alternatively, you can write:

```
secondDay = daysOfWeek[2]
put secondDay
-- "Mon"
```

As you might expect, when using the `getAt()` function, you have to be careful about specifying the position within the list from which you want to get data. If you request a position that's greater than the number of items the list holds, as in the following example, Director displays the Alert message shown in Figure 12-1:

```
eighthDay = daysOfWeek[8]
```

Figure 12-1: Director complains if the requested position exceeds the number of items in the list.

The brackets make life much easier when you are dealing with lists within lists. Suppose that you had the following list:

```
myTable = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

You want to get the third item of the second list. If you think of it as a table, then you want to retrieve the item in row 2, column 3. You need to write the following:

```
put getAt(getAt(myTable, 2), 3)
-- 23
```

Yipes! Gets confusing to look at, doesn't it? The inner function is executed first, and it returns the second list. It's like writing:

```
put getat(myTable, 2)
-- [21, 22, 23]
```

Then the outer function executes on the return value, which is the second list, [21, 22, 23]. It's like writing:

```
put getAt([21, 22, 23],3)
-- 23
```

You could make life simpler by doing it in steps, using a temporary variable:

```
put getAt(myTable, 2) into temp
put getAt(temp, 3)
--23
```

That's much easier to read.

With brackets, it is even easier and involves less typing, as follows:

```
put myTable[2][3]
-- 23
```

The first index holds the row index, and the second holds the column index.

getLast()

This handy function gets the last item from a list.

Syntax:

```
getLast(myList)
myList.getLast()
```

Now try it on the `daysOfWeek` list:

```
put getLast(daysOfWeek)
-- "Sat"
put daysOfWeek.getLast()
-- "Sat"
```

getOne()

This function returns the position of an item in a list, and it works with both linear and property lists.

Syntax:

```
getOne(myList, aValue)
myList.getOne(aValue)
```

myList is the name of your list or the variable holding a list. *aValue* is a value you expect to be in the list. Perhaps, however, you are testing the list to see if that value is in it. If the list does not contain the value, it returns 0.

```
put getOne(daysOfWeek, "Tue")
-- 3
put getOne(daysOfWeek, "Tuesday")
-- 0
put daysOfWeek.getOne("Sat")
-- 7
```

getPos()

This function works in the same way as the `getOne()` function.

Syntax:

```
getPos(myList, aValue)
myList.getPos(aValue)
```

You get the same results as `getOne()`:

```
put getPos(daysOfWeek, "Tue")
-- 3
put getPos(daysOfWeek, "Tuesday")
-- 0
put daysOfWeek.getPos("Sat")
-- 7
```

Setting values

You set values in a list almost as often as you get values from one. Setting a value just means changing a value at a particular location to a new value.

If you want to change an entry in a list, the simplest way is to use the `setAt` command, which mirrors the `getAt` command.

Syntax:

```
setAt myList, index, aValue
myList.setAt(index, aValue)
myList[index] = aValue
```

The `setAt` command passes the name of the list, the position that you want to change, and the value you want to change as arguments. As an example, from a collection of pets, you can create a list with the following animals:

```
pets = ["dog", "cat", "fish"]
```

Here's how to change the third entry ("fish") to a bird and then a hamster, using the `setAt` command:

```
setAt pets, 3, "bird"
put pets
-- ["dog", "cat", "bird"]
pets.setAt(3, "hamster")
put pets
-- ["dog", "cat", "hamster"]
```

Note that you did not have to put parentheses around the `setAt` arguments. This is because `setAt` is a command, not a function. It does not return a value, so the parentheses are optional. We like to add them to improve the readability of the code. When using the dot notation, `pets.setAt(3, "newt")`, the parentheses are required. If you can't remember whether the arguments need to be in parentheses, then use them (it won't hurt and it might make your code easier to understand).

```
setAt(pets, 3, "newt") -- parentheses not required, but
nice
put pets
-- ["dog", "cat", "newt"]
pets.setAt(3, "newt")  -- parentheses ARE required here
```

Be careful about passing the `setAt` command a number outside of its range. The results may surprise you:

```
setAt(pets, 7, "snake")
put pets
-- ["dog", "cat", "newt", 0, 0, 0, "snake"]
```

When the index to the `setAt` command falls outside the range of the list, Lingo pads the list, inserting zeroes for items that have not been defined. It does this with the list operators as well. Although this approach may be useful in some (unusual) circumstances, in general you should avoid it, because it can introduce spurious data into your lists.

As with the `getAt()` function, you can also use the list operators, `[ ]`, instead of the `setAt` command, as follows:

```
pets = ["dog", "cat", "hamster"]
pets[1] = "iguana"
put pets
-- ["iguana", "cat", "newt"]
```

Adding to a list

Sometimes you don't want to change a value in a list, but you want to add additional values. You can add to a specific location in a list or just tack it onto the end.

addAt

The `addAt` handler works similarly to `setAt`. Whereas `setAt` replaces the value at a given position, `addAt` inserts the value at that position, shifting all subsequent items back one position. The `addAt` handler works only with linear lists.

Syntax:

```
addAt myList, index, aValue
myList.addAt(index, aValue)
```

Continuing with the pets example:

```
pets = ["fish", "dog", "cat", "hamster"]
addAt pets, 3, "bird"
put mylist
-- ["fish", "dog", "bird", "cat", "hamster"]
```

Note how "bird" was put into position 3, and the other elements after it are still there. This is the difference between adding an element to a specific position in a list and setting an element in a specific position in the list to a new value.

Using the dot operator, you can add a newt to the first position and bump all the lesser pets back:

```
pets.addAt(1, "newt")
put pets
-- ["newt", "fish", "dog", "bird", "cat", "hamster"]
```

As with `setAt`, if you specify an index position outside of the range of the list with `addAt`, Lingo pads the list with zeroes to fill the undefined locations in the list.

```
pets = ["fish", "dog"]
addAt pets, 5, "newt"
put pets
-- ["fish", "dog", 0, 0, "newt"]
```

If a list has been sorted, using `addAt` breaks the sorting.

```
pets = ["dog", "cat"]
sort pets
put pets
-- ["cat", "dog"]
addAt pets, 2, "bird"
put pets
-- ["cat", "bird", "dog"]
```

add

You can add an item to a list by way of the `add` command. The `add` command places the item at the end of the list, unless the list is sorted. You can use it only with linear lists.

Syntax:

```
add myList, aValue
myList.add(aValue)
```

In the Message window, type the following code:

```
pets = []
add pets, "dog"
add pets, "cat"
put pets
-- ["dog", "cat"]
```

Using `add` is straightforward. If you want a quick list of the names of all the members in a `castLib`, for example, you can write a function like the one shown in Figure 12-2. `GetMemName()` also includes some commented code in it that shows how the statement inside the repeat loop could be written using the dot notation syntax.

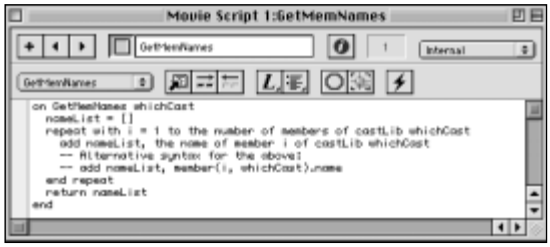

Figure 12-2: A function using add to build a list of member names

It becomes a little less straightforward after you've used the `sort` command on it. If the list is sorted, then additions to the list are added and put in their sorted position automatically, as follows:

```
sort pets
put pets
-- ["cat", "dog"]
add pets, "bird"
put pets
-- ["bird", "cat", "dog"]
```

Because "bird" was added to a list that had been sorted, it was not put at the end of the list but in the right alphabetical position.

append

The append command always puts the value at the end of the list, even if it is sorted.

Syntax:

```
append myList, aValue
myList.append(aValue)
```

Continuing with pets from the preceding example:

```
append pets, "alligator"
put pets
-- ["bird", "cat", "dog", "alligator"]
pets.append("iguana")
put pets
-- ["bird", "cat", "dog", "alligator", "iguana"]
```

In the pets example, the items are no longer in sorted order when the append command is used.

If a list has been sorted, using the append command breaks the sorting.

Deleting items from linear lists

You can also remove values from lists. Linear lists have two commands for deleting items, deleteAt and deleteOne.

deleteAt

To remove a specific item from a linear list, use the deleteAt command. This command eliminates the item at a specified position and then collapses the list by one.

Syntax:

```
deleteAt myList, index
```

To delete the second item in this list, write the following code:

```
pets = ["cat","dog","fish","hamster"]
deleteAt pets, 2
put pets
-- ["cat", "fish", "hamster"]
```

deleteOne

If you want to remove something from a list based on its value, you can use deleteOne.

Syntax:

```
deleteOne myList, aValue
```

For example:

```
pets = ["cat", "dog", "fish", "hamster"]
deleteOne pets, "cat"
put pets
-- ["dog", "fish", "hamster"]
```

If there are two items in the list with the same value, the first item is deleted.

```
pets = ["cats", "dogs", "cats"]
deleteOne pets, "cats"
put pets
-- ["dogs", "cats"]
deleteOne pets, "cats"
put pets
-- ["dogs"]
```

Lingo doesn't have a command that enables you to remove all instances of a value, so this would be a good time to create one by writing our own handler in Lingo (see Figure 12-3).

Figure 12-3: A handler that removes all instances of a value in a list

`RemoveAll` takes advantage of `getOne()` and `deleteOne`. First, `getOne()` checks whether `anItem` is in the list, and if it is, `deleteOne` removes it. The handler keeps going through this loop until `getOne()` returns 0, at which point the loop exits. This handler is not a function; it does not return a value.

```
pets = ["cats", "dogs", "cats"]
RemoveAll pets, "cats"
put pets
-- ["dogs"]
```

So how does the modified list get back to where you called `RemoveAll` from? See the sidebar "Transferring and Duplicating Lists."

Transferring and duplicating lists

When you assign a variable holding a basic data type, such as an integer, float, symbol, or string, to another variable, that value is copied to the other variable. That value exists in a different chunk of memory. The following statements and their results should come as no surprise to you:

```
intA = 12
intB = intA
put intB
-- 12
intB = 7   -- we'll change intB
put intB
-- 7
put intA   -- intA is unaffected
-- 12
```

A list is kept internally as a pointer to a structure in memory; if you assign one variable (which contains a list) to another variable, then the only thing transferred is the pointer, *not the list itself.* In other words, if you set `listB` to equal `listA` and change the contents of `listB` (or `listA`), the changes are reflected in both lists. That's because both variables point to the same list. This can best be demonstrated in the following example:

```
listA = [1, 2, 3, 4, 5]
listB = listA
add listB, 6          -- we'll change listB
put listB
-- [1, 2, 3, 4, 5, 6]
put listA
-- [1, 2, 3, 4, 5, 6] -- They're the SAME!
```

This may not be the behavior that you desire if you want to create a new list containing all the components of the old list and be able to change the new list without affecting the original. With the `duplicate` function, you have the capability to create a copy of a list and modify it separately from the list upon which it was based. The `duplicate` function copies the entire list into a new block of memory and then assigns a new pointer to that block:

```
listA = [1,2,3,4,5]
listB = duplicate(listA)
add listB, 6
put listB
-- [1, 2, 3, 4, 5, 6]
put listA
-- [1, 2, 3, 4, 5]
```

To use `duplicate()` with dot notation, write:

```
listB = listA.duplicate()
```

When you copy a pointer, it happens in an instant. On the other hand, copying the entire contents of a large list (and lists can become *very* large) may take a noticeable amount of time, perhaps even seconds (an eternity!). For this reason, as you design your project, try to minimize the amount of actual list duplication. It can have a profound impact on performance.

When you call a handler with parameters, the values you pass in as arguments are copied into the parameters. So, when a list is passed to a function, as in Figure 12-3, the pointer to the list is what is copied to the parameter. That means the operations are manipulating the same list, not a duplicate. Thus, you do not need a return statement though it wouldn't hurt to have one, and it may make your code clearer to others; see the figure below).

The RemoveAll handler changed to the function RemoveAll().

Here's how you use the altered `RemoveAll()`:

```
pets = ["cats", "dogs", "cats"]
pets = RemoveAll(pets, "cats")
put pets
-- ["dogs"]
```

This snippet of code seems clearer than the `RemoveAll` handler version. Here, it is obvious that the variable `pets` is being changed by the `RemoveAll()` function.

Working with Property Lists

Properties — of sprites, of cast members, of the system itself — form the core of Director Lingo programming. Through Lingo, you can also create your own property lists, thus opening the door for database processing, advanced pro-gramming in behaviors, and object-oriented programming.

A property list includes two related components for each entry in the list. The first component, called a *property*, is linked to a second component, called a *data element*. A property list is like a two-field database. Using this analogy, the first field is the key field (the property) and the second field contains linked data. When you sort a property list, it is ordered by the property.

The syntax for a property list is demonstrated in the following example of test scores. In this example, the names (strings such as "Pat", "Joan", and "Mary Ellen") are properties, and the scores (such as 1200, 1545, and 950) are data. Each name and its associated score is an item in the list.

```
scores = ["Pat": 1200, "Joan": 1545, "Mary Ellen": 950]
```

Strings are not the best choice for property labels. They are slower to process, but more important, using a string as a property label is the only time that you encounter a case-sensitivity issue with Lingo.

Suppose that you tried to get one of the properties and used the wrong case, such as:

```
put getProp(scores, "pat")
```

The result is a syntax error. Most of the time, it is preferable to use symbols, which are not case sensitive.

```
scores = [#Pat: 1200, #Joan: 1545, #MaryEllen: 950]
put getProp(scores, #pat)
-- 1200
```

You can initialize or create an empty property list named Scores by using this syntax:

```
scores = [:]
```

Consider the following situation: You are developing an application in Director that displays a catalog of merchandise. Each "page" of the catalog contains the name of a product, a picture showing the merchandise, a category that describes the type of merchandise, a description of the merchandise, a unique ID, and a price. To make things more complicated, the catalog is updated every week and distributed via the Internet to thousands of customers who don't take kindly to long downloads. In other words, you don't have the luxury of creating a frame-based solution that provides for each page of the catalog having a separate frame, or have a new catalog downloaded each week to all the customers.

Does this sound like an impossible challenge? Actually, it's a scenario typical of many product-related environments, thanks to Internet-fostered changes that make hybrid CD-Web products feasible. Back to the point at hand: This example illustrates the principle of properties in a big way. If you think of a product in the catalog as an object in the same vein as a cast member or a sprite, then the page has several distinct properties. These properties are the nouns that quantify the page, describing the makeup of the page. We might come up with several properties, just from the description of the application, as shown in Table 12-2.

Table 12-2
Properties of the Catalog Product Object

Property	*What It Describes*
The Name of Product	The name of the merchandise
The Picture of Product	A picture (or at least the filename of a picture)
The ID of Product	A unique identifying number for the product to help expedite processing
The Category of Product	A string specifying a given product category (for example, a book might be in the fiction category)
The Description of Product	A text description of the product
The Price of Product	The price of the product (we'll assume this is a floating point number)

Why are these characteristics described as properties of a list, rather than just generic variables? In a catalog, there may be hundreds of different products with various prices, descriptions, names, and so forth — but the important thing is that each item has its own description, name, price, and even maybe an ID.

If you could create a generic product and then customize it to describe uniquely this widget or that gadget, you could make routines that worked with these properties without having to know any specifics about what the properties contain. Again, the principle of separating the functionality of your code from the content of the data comes into play. And, of course, there is a way — in Lingo — to specify such a generic product: That's what the specialized property list is for. Although this list shares some of the same characteristics as a linear list, it also has some considerably expanded functionality.

In the following example, the property list is made up of a collection of entries, each referenced by the property name given as a symbol. For example, the following code segment represents a property list for a paperback book in a catalog application:

```
product = [ #name: "The Novaenglian Chronicles", ¬
            #picture: "NovChron1.jpg", ¬
            #ID: "BKSTR123NC12", ¬
            #category: "Science Fiction", ¬
            #price: 5.95, ù
            #description: "The rise of civilization in post-¬
                           apocalyptic England." ¬
          ]
```

The preceding lines of code constitute one executable statement. The continuation character (¬) enables you to break a long line of code into shorter lines, which can still be considered part of the same statement. When Director evaluates this instruction, it assembles the smaller parts together before interpreting the entire line. To enter the continuation character, press Option+Return (Alt+Return). The continuation character does not work in the Message window.

If you are following along, experimenting in the Message window, then enter product, as follows:

```
product = [#name: "The Novaenglian Chronicles", #picture:
"NovChron1.jpg", #ID: "BKSTR123NC12", #category: "Science
Fiction", #price: 5.95, #description: "The rise of civilization
in post-apocalyptic England."]
```

Each list entry is given a label, using a symbol, which describes the property. Like a linear list, this property list is marked by square brackets. In contrast to a linear list, however, each entry uses a symbol followed by a colon (:).

After you define a property list as just shown, you can refer to the properties of the list as if the list were an object. For example, you can write the following code so that, after the preceding code fragment is encountered, you get the result shown:

```
put "The price of "&(the name of product)&" is $"&(the price of
product)&"."
-- "The price of The Novaenglian Chronicles is $5.95."
```

If `the price of product` property on your system displays as 5.9500, it's because the `floatPrecision` property is set to 4, the default value. This property, which establishes the number of digits that a floating point number displays after the decimal point, can use a value between 0 and 15. Only the display of the float is affected by this property, not the calculation of values. Set the `floatPrecision` to 2 if you want it to display as in the example.

For all intents and purposes, Lingo now treats `product` as an object, in much the same vein as a cast member or sprite. This works in assignments as well. You can alter the price (the book is remaindered) by using the following instruction:

```
set the price of product to 3.95
put "The price of " & (the name of product) & " is $" & (the
price of product) & "."
-- "The price of The Novaenglian Chronicles is $3.95."
```

You can also use dot syntax:

```
product.price = 3.95
put "The price of " & (product.name) & " is $" &
(product.price) & "."
```

You can even the list operators:

```
product[#price] = 3.95
put "The price of " & (product[#name]) & " is $" &
(product[#price]) & "."
```

Note the differing syntax. When you use the property and dot syntax, the # is left off the name of the symbol. Were you to leave it on, you'd get a syntax error. If you left the pound sign (#) off when using the list operators `[ ]`, Director would think you are using a variable.

The use of symbols as labels in a property list is not required — you can use numbers, strings, and even lists. If you don't use symbols, however, you cannot use the property syntax (`the price of product`) or the dot syntax, and several other operations with property lists become less efficient. If you use strings as labels, you have to deal with case-sensitivity. Unless you have a compelling reason to do otherwise, stick with symbols as property labels.

Although property lists can be viewed similarly to objects, they also have many of the same characteristics as linear lists — to the extent that a property list is like a "superset" of linear lists. Property lists give you more ways to get and set data than linear lists do.

Both linear and property lists share a common set of commands and functions. In general, when a linear list function is used on a property list, it acts on the property list's values (data elements), not on the property labels. Using the preceding example, the command `getAt(product1, 4)` retrieves the data associated with the fourth entry in the product list (the value: "Science Fiction") — as opposed to the property name of the fourth entry (`#category`). The primary benefit of using a property list is that you don't need to know a given property's position in the list. You need to know only that the property list contains the property. Don't waste the primary benefit of a property list — avoid relying on positional references for accessing entries unless you are retrieving all of the properties from the list.

The `sort` command sorts the *properties* alphabetically, instead of sorting the data elements.

```
set myList to [#b:3, #a:2, #c:1]
put myList
-- [#b: 3, #a: 2, #c: 1]
sort myList
put myList
-- [#a: 2, #b: 3, #c: 1]
```

This command is the only exception to the rule that functions common to linear and property lists work exclusively on the entries of the list (the data elements), not on the properties. This idiosyncrasy can be a source of errors, because it may seem more logical to sort by the values of the properties.

Many list functions/commands work for both property lists and linear lists. Where a function/command works identically for property lists as it does for linear lists, we'll refer you back to the section on linear lists.

Getting values from property lists

You can get a property from a list by using the `propName` of `myList` syntax, as shown earlier.

getProp()

This function returns the value of the property. If the property does not exist, it returns a syntax error.

Syntax:

```
getProp(myList, propName)
myList.getProp(propName)
```

To use `getProp()` just pass in the name of a property that is in the list:

```
membList = [#happy: (member 1 of castLib 1), #sad: (member 2 of
castLib 1)]
put getProp(membList, #happy)
-- (member 1 of castLib 1)
put membList.getProp(#sad)
-- (member 2 of castLib 1)
```

dot

As in the case of `setProp` and the dot operator's similar functionality, `getProp` and the dot operator work in similar ways. The property must be there for it to work. It is more limited in use because, to use the dot operator, the property labels must be of the symbol data type. It does not work if the property labels are strings or integers.

Syntax:

```
myList.propName
```

Because we used symbols in the `getProp()` example, we can rewrite it using dot notation.

```
membList = [#happy: (member 1 of castLib 1), #sad: (member 2 of
castLib 1)]
put membList.happy
-- (member 1 of castLib 1)
```

getAProp()

This function returns the value of the property. If the property does not exist, it returns <Void>. This can be useful if you deal with a void return value in your code.

Syntax:

```
getAProp(myList, propName)
myList.getAProp(propName)
```

Use getAProp() the same way as you would getProp(), or take advantage of the <Void> return value.

```
on GoSomewhere aDest
  choices = [#Japan: 5, #China: 10, #HongKong: 15]
  myDestination = getAProp(choices, aDest)
  if myDestination = VOID then
    go to the frame
  else
    go to frame myDestination
  end if
end
```

list operators—[ ]

Just as setAProp and [] function similarly, so do getAProp() and []. The brackets return a property that is in the list or void if it does not exist.

Syntax:

```
myList[propName]
```

The GoSomewhere handler in the getAProp() example can be rewritten using brackets to achieve the same functionality.

```
on GoSomewhere aDest
  choices = [#Japan: 5, #China: 10, #HongKong: 15]
  myDestination = choices[aDest]
  if myDestination = VOID then
    go to the frame
  else
    go to frame myDestination
  end if
end
```

The case with symbols

As stated in Chapter 11, the symbol data type in Lingo is not case-sensitive. If you try the following:

```
put #SYMBOL = #symbol
```

the result returns true. Nevertheless, sometimes the case does not appear the way you want it to. For example:

```
set mySym to #Name
put mySym
-- #name
set sym2 to #ASDF
put sym2
-- #ASDF
```

Yipes! What 's going on? Whenever you create a new symbol, your movie forever remembers it the way you first used it. Almost romantic, in a way, but it'll drive the anal-retentives out of their minds. This issue can be serious if you are using the name of your properties in your formatting of output (by using `getPropAt()`).

What can you do about it? Well, if the word you are using already exists in Lingo, such as `name`, then you are out of luck. If you create a word that is not in the Lingo lexicon, then it appears the way you first used it. In the preceding example, #ASDF always appears in all uppercase.

```
put #asdf
-- #ASDF
```

It goes beyond Lingo words and symbols. Suppose that you have a handler called `DumpList`, and you have compiled the scripts, and then you type the following in the Message window:

```
put #dumplist
-- #DumpList
```

The case is the same as the handler's. In fact, any variable name, local or global, sets the case. After you create a variable or handler, for example, you have set the case for the word. The only time this becomes obvious is in the case of symbols, because you can output them.

Is there anything you can do—say, if you accidentally typed the following:

```
put #jOhn
-- #jOhn
put #John
-- #jOhn
```

We admit it: We're retentive types who had to figure out a way around this issue. If it is driving you nuts, too, you'll be happy to know that there is a way to reset these values (except for the words that are part of the Lingo language). Here's what we discovered that works:

1. Comment out all your scripts.
2. Save and compact.
3. Quit.
4. Relaunch.
5. Uncomment all your scripts.
6. Recompile all scripts.

Getting property information

As mentioned earlier, it's not generally recommended that you use positional information to retrieve entries in a property list. Nevertheless, sometimes you need a list of the properties in a property list. Because the `repeat with...in` syntax pulls out only values, not properties, you actually have to request the property at a specific position in the list in order to retrieve the property name.

The `getPropAt` command takes two parameters: the property list name and the position within the list. It returns the property and associated data element located at that position. If you want to output all of the properties and their associated values to the Message window (frequently a useful technique for debugging problems in property lists), here is the `dumpList` handler to use for that task.

You can use this command to get a "dump" of the contents of the product list in the Message window:

```
product = [#name: "The Novaenglian Chronicles", #picture:
"NovChron1.jpg", #ID: "BKSTR123NC12", #category: "Science
Fiction", #price: 5.95, #description: "The rise of civilization
in post-apocalyptic England."]

DumpList product
-- "name: The Novaenglian Chronicles"
-- "picture: NovChron1.jpg"
-- "ID: BKSTR123NC12"
-- "category: Science Fiction"
-- "Price: 5.9500"
-- "Description: The rise of civilization in post-apocalyptic
England."
```

If you typed the property #name in a different case — for example, #NAME — and you are wondering why it appears as #name, see the earlier sidebar "The Case with Symbols."

You might wonder why the DumpList handler doesn't just reference the data from its position, rather than from the property. Surprisingly, the answer is because it's faster to do it from the property. The entries are internally referenced by property, and if we used the position, Lingo would have had to find the property corresponding to that position, and then retrieve the entry from the property. By using the property to get the data directly, we cut down on a step that Director would have done otherwise.

Setting values, revisited

You can set a property by using the `set the propertyName of myList to aValue`. There are many other ways to set property values.

setAt

This command works identically with both linear and property lists. See the discussion of `setAt` under "Setting Values" in the "Working with Linear Lists" section of this chapter.

setProp

This command takes the name of the property to which you are giving a new value and sets it to that value. It causes an error if the property does not exist in the list.

Syntax:

```
setProp myList, propName, aValue
myList.setProp(propName, aValue)
```

The `setProp` command sets a property's value to a new value:

```
product = [#name: "Director", #Version: 7.0]
setProp product, #Version, 8.0
product.setProp(#Version, 8.0)
```

dots

Using the dot operator works like the `setProp` command: The property must be there for it to work. It is more limited in use because, in order to use the dot operator, the property labels must be of the symbol data type. It doesn't work if the property labels are strings or integers.

Syntax:

```
myList.propName = aValue
```

Rewriting the setProp example, using the dot example, can save some keystrokes. When you use this format, don't put the # sign in front the of the symbol name.

```
product = [#name: "Director", #Version: 7.0]
product.Version = 8.0
put product.Version
-- 8.0
```

setAProp

This command takes the name of the property to which you are giving a new value and sets it to that value. It adds the property to the list if it does not exist.

Syntax:

```
setAProp myList, propName, aValue
myList.setAProp(propName, aValue)
```

You can use this command to set existing properties, or you can use it to add ones that do not exist:

```
product = [#name:"Freehand"]
setAProp product, #name, "Director"
setAProp product, #Version, 8.0
put product
-- [#name: "Director", #Version: 8.0]
```

Because an addProp command does exist, we don't recommend using setAProp to create new properties. Someone who comes along later to modify your code might not understand that you are intentionally using it this way. It is clearer to use addProp when you want to add a property and setAProp (or setProp) when you want to set existing properties.

If you accidentally misspell the name of a property when using setProp, Director alerts you that something is wrong. If you use setAProp instead, then the misspelled property name is blithely added to the list and you are none the wiser until the misspelling generates a syntax error down the line or a value is not properly updated. Unless you have a strong reason for using the setAProp routine to create a property, you should use setProp instead.

[ ]—List Operators

You can use the list operators [] with property lists as well as with linear lists. Where you would use an index with a linear list, you use a property name or an index. If the property does not exist in the list, it acts like the setAProp command, creating the property.

Syntax:

```
myList[#propNameOrIndex] = aValue
```

We can rewrite the setAProp example to use brackets. Like the dot operator, list operators save us some keystrokes:

```
product = [#name:"Freehand"]
product[#name] = "Director"
product[#Version] = 8.0
put product
-- [#name: "Director", #Version: 8.0]
```

Adding an item to a property list

In order to add an item to a list, use the addProp handler, passing to it the list, the property name (preferably as a symbol), and the entry.

Syntax:

```
addProp myPropList, propName, aValue
myPropList[propName] = aValue
myPropList = [:]
addProp myList, #Version, 8.0
addProp myList, #name, "Director"
put myList
-- [#Version: 8.0, #name: "Director"]
```

You can also do it this way:

```
myList = [:]
myList[#Version] = 8.0
myList[#name] = "Director"
put myList
-- [#Version: 8.0, #name: "Director"]
```

If the put myList instruction displays this result

```
-- [#Version: 8.0000, #name:"Director"]
```

(with four digits after the decimal point), issue the set the floatPrecision to 2 instruction, and then issue the put myList instruction again. The new result is

```
-- [#Version: 8.0, #name:"Director"]
```

Deleting properties

With property lists, you have two options for deleting a property. You can delete by using the name of the property with the deleteProp command, or you can delete by index, using the deleteAt command.

deleteProp

This command deletes an item from a list. If the property does not exist, you do not get an error.

Syntax:

```
deleteProp(myList, propName)
myList.deleteProp(propName)
```

If you want to delete the category property from a product, for example, you can use the following syntax:

```
put the category of product
-- "Science Fiction"
deleteProp product, #category
put the category of product
-- This puts up an alert box "Script Error: Property not found"
```

deleteAt

This command works identically with both linear and property lists. See the discussion of `deleteAt` under "Deleting Items from Linear Lists" in the "Working with Linear Lists" section of this chapter.

Searching a dictionary list

The property list has a complementary concept in other programming languages: the dictionary. With the dictionary approach, each property of an object or list is like a term in a dictionary. Properties equate to the words that are defined in a dictionary, and data elements equate to the definitions of those words in a dictionary. It is probably with this view in mind that the engineers at Macromedia created the `findPos()` and `findPosNear()` functions.

findPos

The `findPos()` function requires two parameters: a property list and a symbol. It returns the position of that property within the list. If it doesn't find the property, then the function returns a value of `<Void>`.

Syntax:

```
findPos(myList, aValueOrProp)
```

This function is analogous to the `getPos()` function for entries. Here is `findPos()` at work:

```
set myList to ["lions", "tigers", "bears"]
put findPos(myList, "tigers")
```

```
-- 2  -- "tiger" is the second item in myList
put findPos(myList, "cats")
-- 0  -- "cats" is not found so 0 is returned
```

Now look at `findPos()` in a property list:

```
set myList to [#c: 33, #a: 82, #d: 64]
put findPos(myList, #c)
-- 1   -- #c is indeed in position 1
put findPos(myList, 64)
-- <Void>   -- returns <Void> because 64 is not a property in
               myList (even though it is a value at a property)
```

If you wanted to find the value 64 in the preceding property list, you could use the `getPos()` function. It does work based on value whether you are using it on a linear or property list.

findPosNear

The `findPosNear()` function is a little more intriguing. In a sorted list, `findPosNear()` compares the passed symbol with all of the other symbols, and indicates what position the passed symbol would take if it were sorted into the list. The result implies that the property currently occupying that position would be the closest "match" to the passed symbol. This works especially well with "type-ahead" projects, such as a Help File Index, where the user of the product types in a string expression and the program returns the closest match.

Syntax:

```
findPosNear(myList, aValueOrProp)
```

To use `findPosNear()`, you must first sort the property list. If the property list is not sorted, both functions return 0 for linear lists and the last position for property lists.

Here's the preceding example, using `findPosNear()`:

```
set myList to ["lions", "tigers", "bears"]
put findPosNear(myList, "cats")
-- 0   -- returns 0 in unsorted list
sort myList
put myList
-- ["bears", "lions", "tigers"]
put findPosNear(myList, "cats")
-- 2  -- returns correct position in sorted list
```

When you use `findPosNear` on `myList`, and pass in "cats", it returns 2. Why? Because if you were to add "cats" to `myList`, it would fall between "bears" and "lions"—in the second position.

And in a property list:

```
set myList to [#c: 33, #a: 82, #d: 64]
put findPosNear(myList, #b)
-- 4   -- last position because list is unsorted
sort myList  -- sorts by property, not value at property
put myList
-- [#a: 82, #c: 33, #d: 64]
put findPosNear(myList, #b)
-- 2   -- in sorted list it returns correct position.
```

Here's a good benchmark for deciding whether to use linear lists or property lists: If the positions of the items within the object are more important, as they would be in a trivia contest or card game, then use linear lists. If the characteristics of the object—the object's name or description, for example—are more important, then your object should probably be defined as a property list.

Processing with Lists

Lists can take a certain amount of effort to understand, but the work is generally worth it. Lists play a big part in any kind of database design, and many of the more useful properties in Director (such as `the actorList`, `the windowList`, and `the scriptInstanceList`) are linear lists.

Lists are processed much more quickly than strings. They can be used to make lists that shadow text blocks; these can then be processed and transferred back to strings or text field entries. In this section, we demonstrate several useful routines for converting, processing, and otherwise manipulating lists.

Counting lists

By counting in advance the total number of items in the list, you can avoid "index out of range" errors in functions such as `getAt()`. Let's take a look at the `count()` and other list functions.

Using the `count()` function, you can restrict the `getAt()` command to a value that is not less than 1 (item 1 in the list) and that does not exceed the total number of items in the list, thereby avoiding the ugly alert box. The function shown in Figure 12-4 returns the name of a specific day of the week, contained in the parameter `whichDay`, based on its position in the list. The handler effectively limits the value used by the `getAt()` function to a valid range (no more than the actual number of entries in the list). In this example, the `getDaysOfWeek` routine defines the `daysOfWeek` list and then checks to make sure that the day entered (`whichDay`) falls between 1 and the `count` of the number of items in the list, in this case 7. If it doesn't, the routine displays a dialog box reflecting the bad day (and yes, we've had bad days like that, too) and then quits the program. Otherwise, the correct day of the week is returned.

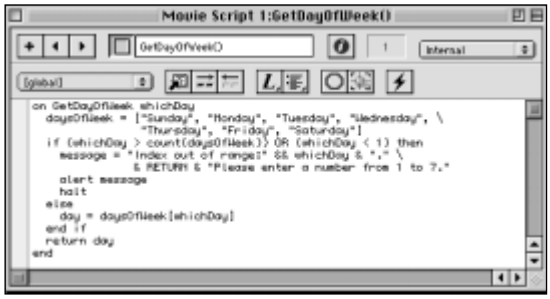

Figure 12-4: Using the count() function in a handler

You might wonder whether you are able to use this same technique using chunk expressions and items in a text string. The answer is yes you can, quite easily. String operations, however, are typically slower than the equivalent list functions — in some cases, an order of magnitude slower. Moreover, although this can be done with relatively simple text items, the more complex operations would become bogged down in chunk references. Whenever you see an operation that can be handled with text items, think about whether the actions can be transformed into list operations instead.

Converting a list to a string

With lists, the `string()` function can convert even a very large and complex list structure into a string, making it useful for intermediate-term storage of lists beyond the scope of the movie. The converted information retains a certain amount of type information — for example, if you have a list of symbols, the function retains the symbol notation:

```
daysOfWeek = [#Sun, #Mon, #Tue, #Wed, #Thu, #Fri, #Sat]
put string(daysOfWeek)
-- "[#Sun, #Mon, #Tue, #Wed, #Thu, #Fri, #Sat]"
```

The `string()` function performs a little magic on lists that contain strings. The function retains the quotation marks inside the list string, and the quotation marks, therefore, don't wind up breaking up the string and generating an error:

```
pets = ["cat", "dog", "fish", "hamster"]
put string(pets)
-- "["cat", "dog", "fish", "hamster"]"
```

The `value()` function works in reverse of the `string()` — if you have a string in the shape of a list, the `value()` function converts it back into a list. This function can be very handy for storing list data, as long as the list data itself is relatively simple (that is, no lists within lists or objects within lists).

```
pets = ["cat", "dog", "fish", "hamster"]
petString = string(pets)
put string(pets)
-- "["cat", "dog", "fish", "hamster"]"
put value(petString)
-- ["cat", "dog", "fish", "hamster"]
```

You can get away with storing lists within lists as strings, but it is a less reliable operation. Lists that should convert to strings and back to lists occasionally become corrupted, while other seemingly improbable candidates go through the conversion process just fine. In general, you're better off saving the data in some other format and then loading the lists later, rather than attempting to save a string-converted list for reconstitution later.

Remember that when you convert lists to strings in this manner, the outcome may not be quite what you were hoping for. We have frequently found it necessary to take a multiple-line or multiple-item selection in a text member or field and convert it into a list, or vice versa. Although Director has no explicit routines to do this, writing your own is simple.

Converting items to a list

You can use a scrolling field to create a *list box*. A list box is just a list of items on the screen; users can scroll through the list and select the item they desire. The text in the list box should not wrap. To implement a list box, your tasks include a lot of list-type activities, such as:

- ✦ Inserting items into a list
- ✦ Retrieving an item at a specific position
- ✦ Finding the position of an item in the list

We need to create a list that corresponds to the list of items in the field. The first item in the list will correspond to the first item in the field, the second item in the list corresponds to the second item in the field, and so on. Using and manipulating the list is much faster than using chunk expressions to manipulate a string, and you have flexibility that is unavailable using only string functions.

The routines `TextToList()`, `GetFolderList()`, `ListToText()`, and `AddToSortedText` are available on the CD-ROM for this book in the CHAPTER 12 folder. The LISTTOTEXT2.DIR file has the finished routines.

Two functions that are indispensable when working with shadow lists are the `TextToList()` and `ListToText()` functions.

The TextToList() function, shown in Figure 12-5, takes as parameters a string or text field reference and an optional delimiter. The delimiter is included so that you can pull data out of a sequence of items as well as from lines of text. Normally, the routine defaults to the system item delimiter, typically a comma. You can convert lines of text into items in a list by passing RETURN as the delimiter. As Figure 12-5 shows, the code for the handler is relatively simple.

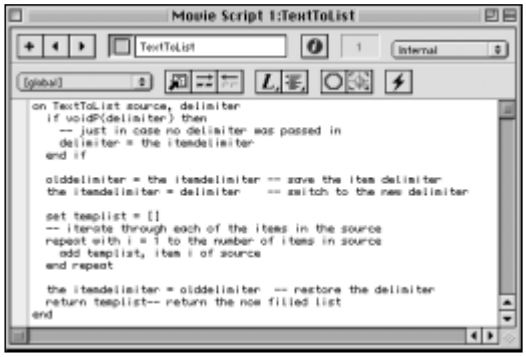

Figure 12-5: The TextToList() function

Almost all the code in the TextToList() function is devoted to making sure that the right delimiter is used. The code that reads items from a string or text field and adds them to a list takes up maybe five lines. Thus, to convert a sequence of items — say, animals at a pet store — into a list, all you need to do is to write the following code:

```
strAnimals = "fish,cat,dog,hamster,snake,bird"
listAnimals = TextToList(myAnimalStr, ",")
put listAnimals
-- ["fish", "cat", "dog", "hamster", "snake", "bird"]
```

As a more practical example, you can convert a file path into a list of folders (see Figure 12-6).

Figure 12-6: The GetFolderList function

You can test this function in the Message window by using the following instructions (minus the lines of comments):

```
-- The following is a Macintosh path because it uses colons as
a path delimiter.
myPath = "Hard Drive:Documents:Myfile"
put GetFolderList (myPath)
-- ["Hard Drive", "Documents", "Myfile"]
```

Alternatively, you can test it with one of Lingo's functions or properties that evaluate to a path, such as `the applicationPath`:

```
put GetFolderList(the applicationPath)
-- ["Macintosh HD", "Director 8", ""]
```

Converting a list to items

The inverse function, `ListToText()`, is a little more complex (see Figure 12-7). Because it is often used to output a list as a series of lines, the `RETURN` character is the default delimiter.

Figure 12-7: The ListToText() function

What we are doing in the repeat loop is adding the entry to a buffer, and then adding the delimiter. This works fine until the last entry: The disadvantage of the `repeat with ...in` loop is that there is no way to know when you have reached the last entry without comparing the entry, and that can add time to the routine. As a consequence, after all the entries have been added to the buffer, we remove the last delimiter from the end of the buffer, and then output the buffer as a string.

Return to the list box concept. The following lines of code might be found in a button that reads the contents of one text box, "Name", and then adds the contents, sorted, into the list of a second text box, "AllNames" (see Figure 12-8).

Figure 12-8: Button behavior that uses both TextToList() and ListToText()

The sort algorithm that Director provides for lists is much faster than anything you could write to process text in Lingo — so much so that the time to convert to a list, sort the list, and convert back to text is negligible in comparison. Indeed, this particular routine is useful enough to convert to a standard handler (see Figure 12-9).

Figure 12-9: Code from mouseDown (in Figure 12-8) turned into a standard handler.

The routine is then called from a `mouseDown` or `mouseUp` script of a button sprite with the field names as parameters. Figure 12-10 shows the routine being called from a `mouseDown`.

Figure 12-10: AddToSortedText being used in mouseDown handler

There is no reason why the delimiter in the `ListToText()` function has to be a single character. If you want to output a list as a series of paragraphs with a line between each paragraph, call the function using this instruction:

```
ListToText(myList, RETURN&"--------"&RETURN)
```

Arithmetic with lists

You can use the arithmetic operators (+, -, *, /) on lists as well. This feature is one of the cooler (though less intuitive) things built into Lingo:

```
set list1 to [2, 2]
put list1 + 3
-- [5, 5]
put list1 * 4
-- [8, 8]
put list1 * .6
-- [1.2000, 1.2000]
put list1/0.5
-- [4.0000, 4.0000]
```

And you can even perform arithmetic on multiple lists:

```
set list1 to [2, 2]
set list2 to [3, 3]
put list1 + list2
-- [5, 5]
put list1 - list2
-- [-1, -1]
put list1 * list2
-- [6, 6]
```

This holds true for property lists as well:

```
set propList1 to [#a: 1, #b:2]
put propList1 * 2
-- [#a: 2, #b: 4]
```

It really gets wild when you start adding property lists to linear lists:

```
set list1 to [2, 2]
set propList1 to [#a: 1, #b:2]
put list1 + propList1
-- [[#a: 3, #b: 4], [#a: 3, #b: 4]]
-- now let's change the order of the addition
put propList1 + list1
-- [#a: [3, 3], #b: [4, 4]]
```

Note how the positions of the operands make a difference in the results. Adding a linear list (list1) to a property list (propList1) results in a linear list of property lists. Adding the property list to the linear list results in a property list in which the property values are linear lists! It makes perfect sense after you think about it (and after your headache goes away).

What about adding lists of differing lengths? In this case, the resulting list is truncated:

```
set list1 to [1,2]
set list2 to [1,2,3,4]
put list1 + list2
-- [2, 4]
```

Is it a list?

Sometimes you might need to check whether a variable is a list or not. Because variables in Lingo can hold any type of value, any type of data could be in that variable. You might be expecting a return value of a list and get `<Void>`, an integer, or perhaps a string. . There are two ways to approach testing.

listP()

This function returns 1 if the value it is passed is a list (property or linear) and 0 if it is something else.

Syntax:

```
listP(aValue)
aValue.listP
```

The first version of the syntax is straightforward. The second one you need to be more careful with. If you add parentheses after `aValue.listP`, you get a syntax error. This is not consistent behavior when compared with other list functions, such as `count()`, where it does not matter whether you use parentheses with the dot notation.

The *P* in the `listP()` function probably stands for predicate. In computer science, functions that return true or false are sometimes called predicates. Lingo has several other predicate functions for testing the type of a variable: `integerP()`, `floatP()`, `stringP()`, `objectP()`.

ilk()

The `ilk()` function is useful in determining the type of list a variable holds.

Syntax:

```
ilk(myList)
myList.ilk
ilk(myList, #type)
myList.ilk(#type)
```

The return value is a symbol if only a list is used as an argument. It returns a Boolean value if you pass in a second parameter.

Using the preceding examples:

```
put ilk(list1)
-- #list
put ilk(propList1)
#propList
```

If you want a Boolean test to determine whether a list is a certain type, add an argument to the function:

```
put ilk(list1, #linearList)
-- 1    -- remember, 1 means TRUE
put ilk(list3, #linearList)
-- 0    -- and 0 means FALSE
put ilk(list3, #propList)
-- 1
```

Remember, the number 1 means TRUE and 0 means FALSE.

And you can be a little more general in your testing by just testing whether it is a list, which is an alternative to using `listP()`:

```
put ilk(list3, #list)
-- 1
put ilk(list1, #list)
-- 1
```

Joining two lists

Lingo does not have a "join" or "combine" command. There are times when such a command would be useful. Fortunately, it is easy to write a function that does this (see Figure 12-11).

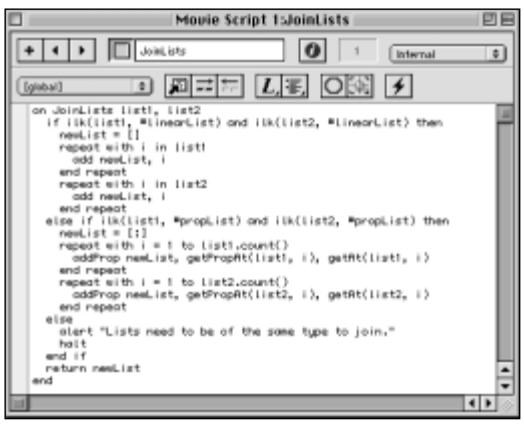

Figure 12-11: The JoinLists() function illustrates how to combine two lists, property or linear.

`JoinLists()` takes two lists as parameters. It tests them to see if they are both the same kind. If they are not, an alert pops up and the program halts. If they are of the same type, then a new, empty list is created. This is why we need to test for the type — to know what kind of list to create. We need to create a new list so that we do not alter list1 or list2. We then return the new, combined list.

Creating a Simple Address Book Program

Lists are perfect for creating small databases. An address book is essentially a database of names and addresses. This program is not full-featured, but more a vehicle to show you how to use lists. After it's complete, you can extend the program if you want; we're sure you'll see many ways to improve it. It also gives you ideas on how to trap user input, instead of keeping track of names and addresses — maybe you are gathering information from users in the form of a test or questionnaire.

You can find the completed addressbook.dir movie on the accompanying CD-ROM in EXERCISE:CH12 (EXERCISE\CH12).

The program features three different screens, each one on its own frame. One is for browsing the data, one is for editing it, and one is for creating new entries. Essentially, the three screens are the same except for the buttons at the bottom. Each screen has a name, street, city, state, and zip field (see Figure 12-12).

Add Browse, New, and Edit markers to frames 1 to 3, respectively.

For each of the fields for input — Name, `Street`, `City`, `State`, and `Zip` — create a text cast member, and draw them right on the Stage so that they occupy sprites 1 through 5. Name each one as we did (see Figure 12-13). Then create another text cast member that holds the bold type. Extend these sprites so that they span all three frames.

Last, add the buttons. To frame Browse, add `Edit`, `New`, `Prev`, and `Next`. On frame New, put `Done` and `Cancel`. On frame Edit, add `Done`, `Cancel`, and `Delete`. It is important that the Done button be a separate sprite in frames New and Edit. Cancel can span both of those frames because it has the same functionality for both.

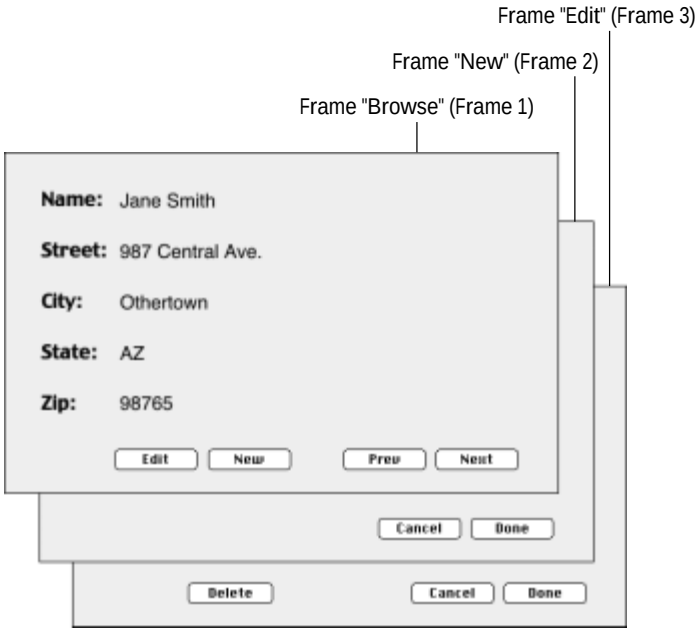

Figure 12-12: All three screens are shown here at the same time. All look the same, except for the bottom buttons. In the program, you view only one screen at a time.

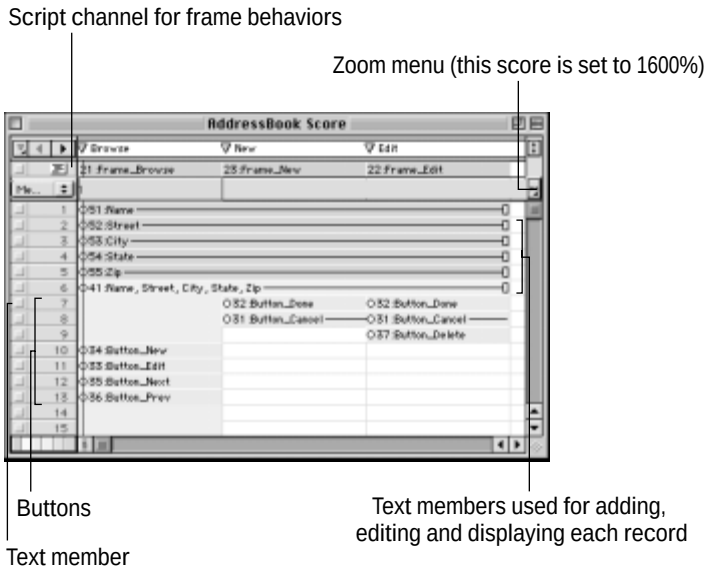

Figure 12-13: The Score of addressbook.dir with the Sprite Toolbar and the effects channels hidden, so that the Score does not take up as much space.

Next, add functionality to the buttons — eight button behaviors in all. The first six are quite simple (see Figure 12-13). Try to keep the code simple so that it can be easily transferred to more complex sprite button behaviors. For this example, you are using the button from the Tool palette. This type of button is good for creating quick prototypes, but graphic designers tend to cringe at their usage. One way to keep the code simple is to have it call a movie script. This enables you to design the program from the top down. The bottom three scripts in Figure 12-14 are calling handlers that you haven't written yet, but it is easy to guess their functionality. It also enables you to easily move the code to another behavior, if necessary; all you have to move is one statement.

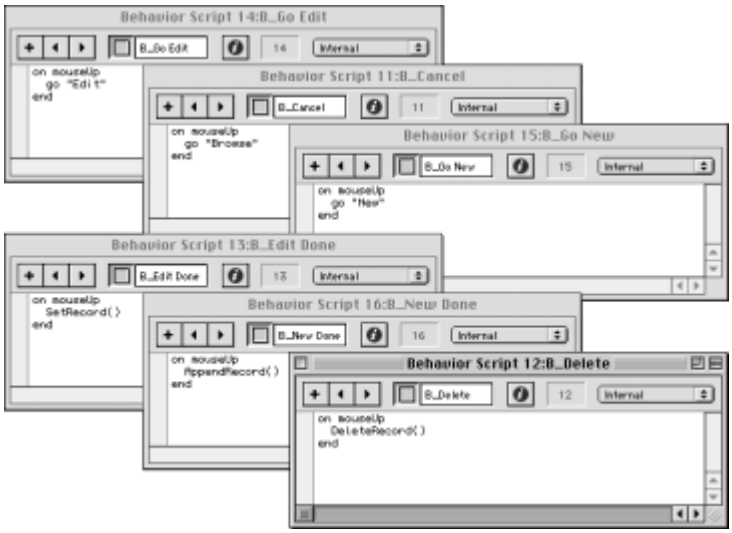

Figure 12-14: Six of the eight sprite behaviors for this movie that have been applied to buttons.

We begin all of the sprite behavior names with the prefix "*B_*", which stands for behaviors. We begin all of the frame behaviors with "Frame:". This works well when we select all of the scripts in the cast window and then sort them by name (Modify➪ Sort). This way, they appear in the Behaviors pop-up menu in alphabetical order and are grouped by how they are used (either on frames or sprites), as shown in Figure 12-15. Another trick is to add two carriage returns at the top of each script window, so that part of the script does not appear in the Behaviors pop-up menu.

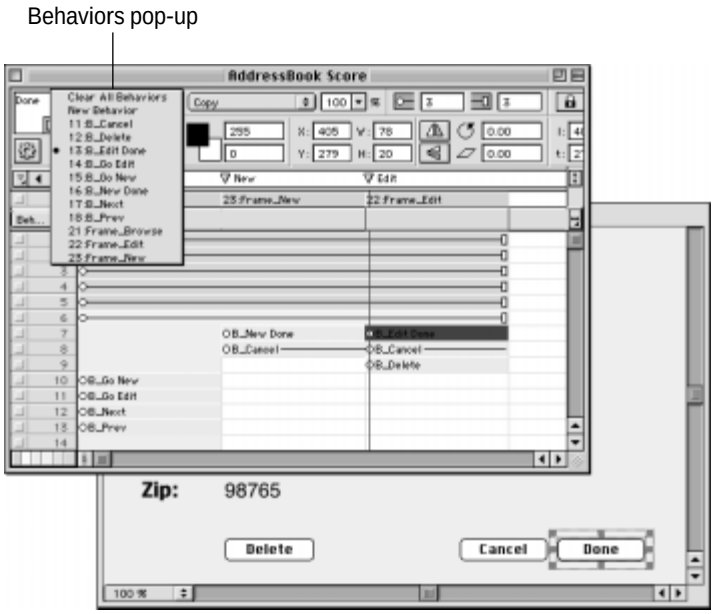

Figure 12-15: By using naming conventions and sorting cast members, you can make your behaviors pop-up menu more readable.

The `Next` and `Prev` button behaviors are a little more complicated, but each is almost identical to the other. They check each time there's an `exitframe` message to see whether they should be visible. If you're on the first record, you don't want the Prev button to be visible. The same is true for the Next button when you are on the last record. On a `mouseUp`, the B_Prev script passes a –1 as an argument to the `ChangeRecord` handler, as shown in Figure 12-16. Think of —1 as going to the left, or previous record. B_Next does the same, except it passes a 1.

The three frame behaviors are shown in Figure 12-17. For the Browse frame, it needs to set the five text cast members to a non-editable state; these members are in sprites 1 to 5. Then it calls `ChangeRecord` and sets it to the current record. Zero is passed in `ChangeRecord`, which means it is not incrementing or decrementing the record you are viewing. The scripts for the Edit and New frames each need to make the text cast members editable. The New frame also needs to clear those members out by putting an empty string in each.

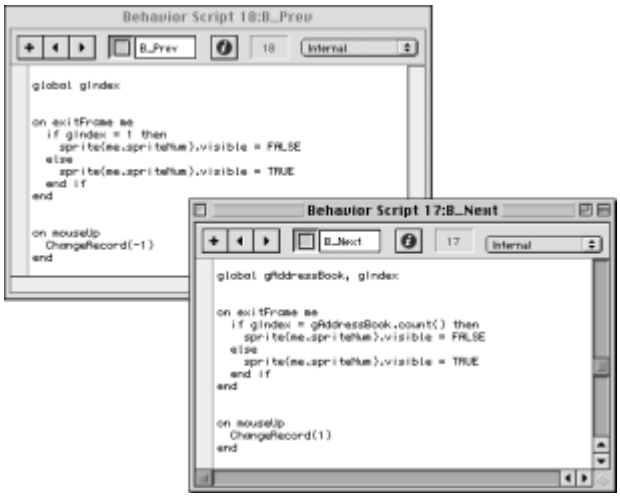

Figure 12-16: The Prev and Next buttons have nearly identical code.

Figure 12-17: The three frame scripts each loop on the frame but they vary in their beginSprite handlers.

Our last task is the Movie scripts. Now we'll finally see how to work lists and list operations into this program! The program starts by checking to see whether `gAddressBook` was initialized, and if it wasn't initialized, it puts some dummy information into the variable. This information comes from the `GetStarterList()` function. `GetStarterList()` returns a linear list that is full of property lists. Each property list represents one record. After that, `gIndex` is initialized to 1. Because we are starting the program, we start on the first record. Figure 12-18 shows what the handlers should look like.

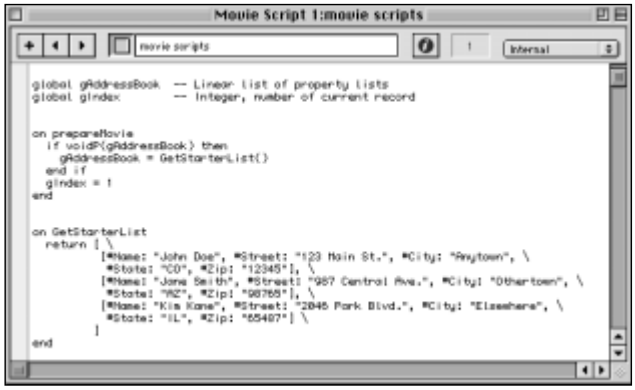

Figure 12-18: The prepareMovie handler checks to see whether gAddressBook was initialized, and if it wasn't initialized, it puts in a few dummy records.

Those aren't all the movie scripts—you have to contend with a few more. The `ChangeRecord` handler is called from three different spots; see Figure 12-19 for the handler's definition. The first thing that happens in `ChangeRecord` is that the global variable `gIndex` is changed by adding the value of the parameter `dir` (short for direction). Then `aRecord` gets a reference to one of the property lists within the linear list `gAddressBook`. Each property list within `gAddressBook` has five elements. We loop five times: the first time setting the text of the member of sprite 1 to the value at position 1 in the property list `aRecord`; the second time setting the text of the member of sprite 2 to the value at position 2 in the property list `aRecord`; and so on through position 5.

Figure 12-19: ChangeRecord fills the text cast members with the information from the current record.

The last four movie handlers to be created are interesting because three of them mimic list commands (see Figure 12-20).

Figure 12-20: Three of these handlers bear names similar to list commands, and their functionality is quite similar.

The `GetFieldInfo()` function gets the value of the five text cast members. We call it `GetFieldInfo()` because the text cast members are being used like fields in a database. Calling it `GetFieldInfo()` seemed like it would be doing something different. This function gets the text and puts it into a property list, with property labels that correspond to the data they hold. Realistically, it could have been made into a linear list, because for this limited program we are referencing values by position. However, making it a property list makes the code more readable, easier to debug, and anticipates expanding the code at a later point, when you may be accessing only certain values from each property list.

The remaining three handlers — `AppendRecord()`, `SetRecord()`, and `DeleteRecord()` — do for our little database what `append`, `setAt`, and `deleteAt` do for lists.

`AppendRecord()` is called from the script B_New Done on the Done button in the New frame. It gets a property list of the new data from `GetFieldInfo()` and puts it at the end of the `gAddressBook` list. Then it increments `gIndex` and goes to the Browse frame.

`SetRecord()` is called from the script B_Edit Done on the Done button in the Edit frame. It gets a property list of the new data from `GetFieldInfo()` and sets that property list on top of the old list that was in the same position. Then it goes to the Browse frame.

`DeleteRecord()` is called from the script B_Delete on the Delete button in the Edit frame. It deletes the current record from `gAddressBook`. It knows which to delete because we have been keeping track of which one we were on by incrementing/decrementing `gIndex`.

That's it! There are many things this example does not do that would be useful. It would be nice, for example, to save information, copy and paste in fields, and check for errors in the last element in the list. Still, this example does illustrate:

- How to iterate through lists
- How to input user information into a list, store it, retrieve it, and delete it
- How to write many small behaviors and handlers
- How to use many of the control structures discussed in the preceding chapter, such as repeat loops, `if...then`, and `if...then else`
- How to navigate Lingo (such as `go`) to jump between frames or stay on the same frame
- How to change text members through Lingo
- How to declare and use global (and local) variables

Points and Rects

Lingo supports two data structures that are common in graphics: points and rects. These two data types also are considered lists by the `ilk()` function and can perform many list operations.

Points

Points are x and y coordinates. The first number in a point is the x coordinate, or locH. The second number is the y coordinate, or locV. As you learn in Chapter 15, locH and locV are sprite properties. They reflect the sprite's location on a two-dimensional plane. An easy way to remember this is *loc* is short for location. *H* and *V* are short for horizontal and vertical, respectively. A point is a data structure that you can put into a variable and manipulate. When you get the loc of a sprite, it returns a point.

Try throwing a sprite on the Stage and type the following in the Message window:

```
put the loc of sprite 1
-- point(246, 126)
```

The point you get is most likely different from the one we got, but you should be able to tell it is returning the location of the sprite. Now put it into a variable:

```
put the loc of sprite 1 into myLoc
put myLoc
-- point(246, 126)
```

The `myLoc` variable now holds a point. It happens to be the loc of sprite 1. If we alter the value of `myLoc`, it does not have any effect on sprite 1 until we actually reset the loc of sprite 1 to `myLoc`. (We'll do more of that in Chapter 15. For now, we are concerned with the data.)

The point data type looks a lot like the list type when you use parentheses instead of brackets. Points are considered both a data type and a function in Lingo. Suppose that you want to create a point based on variable values:

```
set x to 12
set y to 342
set myPoint to point(x, y)
put myPoint
-- point(12, 342)
```

Every point has a locH and locV that can be retrieved by using the property names:

```
put the locH of myPoint
-- 12
put myPoint.locV
-- 342
```

Another common use is to store the location of the mouse. This is useful in conjunction with rects, which you'll see in a moment:

```
 set myLoc to the mouseLoc
```

Because it has properties, you might think you can use `getProp()` on it to retrieve the value. This does not work because it is not considered a property list. You can, however, use `getAt()`, `getPos()`, `getOne`, and `getLast()`:

```
put getAt(myLoc, 1)
-- 246
```

The same holds true for `setAt`:

```
setAt myLoc, 1, 100
put myLoc
-- point(100, 246)
```

As with lists, you can perform arithmetic on them:

```
put myLoc + 10
-- point(110, 256)
```

Rects

A rect (short for rectangle) is a structure that contains four numbers, which represent the left, top, right, and bottom sides of the rectangle. Where points have the properties locH and locV, rects have the properties left, top, right, and bottom. The first number represents the position of the left side of the rectangle, the second is the top, the third is the right, and the fourth number is the bottom.

Syntax:

```
rect(left, top, right, bottom)
rect(point1, point2)
```

Put a sprite on the Stage and then type in the Message window:

```
set myRect to the rect of sprite 1
put myRect
-- rect(114, 75, 238, 199)
```

Rects, like points, are considered both a data type and a function in Lingo. To get or set the individual values, type code like the following:

```
put the right of myRect
-- 476
put the top of myRect
-- 150
```

With rects, you can do arithmetic on them, just as you can with lists:

```
put myRect
-- rect(228, 150, 476, 398)
put myRect + 2
-- rect(230, 152, 478, 400)
put myRect * 3
-- rect(684, 450, 1428, 1194)
put myRect * 1.2
-- rect(273.6000, 180.0000, 571.2000, 477.6000)
put myRect + myRect
-- rect(456, 300, 952, 796)
```

Four numbers can define a rect, but two points can also define a rect. The points do not remain points; they convert to left, top, right, and bottom values.

```
put rect(the loc of sprite 1, the mouseLoc)
-- rect(57, 50, 324, 217)
```

Rects are often used to hold the dimensions of sprites or members. You can use the same list functions that work with points: `getAt()`, `getPos()`, `getOne`, `getLast`, and `setAt`. But rects are not limited to these functions. There are several rect functions:

- `inflate()`
- `inside()`
- `intersect()`
- `map()`
- `offset()`
- `union()`

It is important to remember that a rect is just a structure in memory; you do not see visuals on the Stage when you use them (unless you are setting a sprite's rect to the rect in your variable).

inflate()

You use this function to increase (or decrease) the size of a rect around its center.

Syntax:

```
inflate(myRect, horz, vert)
myRect.inflate(horz, vert)
```

Although rects are just structures in memory, it helps to visualize what happens with these functions.

In Figure 12-21, the gray rectangle represents the original rect and the black rectangle represents the rect returned by the function (after it's been enlarged). The horz and vert values are how much each side moves. So, if you pass a 10 in the horz spot, the overall width of the rect is 20 (the left side moves 10 and the right side moves 10).

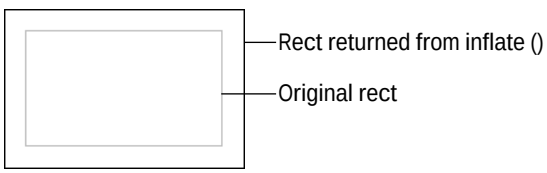

Figure 12-21: Inflating a rect

You might think this is unnecessary because you can do arithmetic on rects. But consider the following:

```
put rect(0, 0, 160, 120) + 10
-- rect(10, 10, 170, 130)
```

You didn't make it bigger, you just moved it 10 to the right and 10 downward. You *could* set each of the individual values to make it larger, but that is unnecessary given this function:

```
put inflate(rect(0,0,160,120), 10, 10)
-- rect(-10, -10, 170, 130)
```

You can also use `inflate()` to shrink rects; just pass in negative values:

```
put inflate(rect(0,0,160,120), -10, -10)
-- rect(10, 10, 150, 110)
```

inside()

We use this function often to create clickable areas in movies where there is no sprite. It checks to see if a point is inside a rect.

Syntax:

```
inside(myPoint, myRect)
myPoint.inside(myRect)
```

Perhaps now that we have up to 1000 sprite channels, we might not use this function as often. Still, it is more memory efficient than using a cast member in a sprite channel. So, if you are aiming for speed or size, this is one way to go. Figure 12-22 provides an example of a frame behavior.

Figure 12-22: Creating a clickable area without a sprite

This example can be expanded to take advantage of your knowledge of repeat loops and lists. Figure 12-23 shows the expanded version (with comments). Now we have a linear list of rects — `myRectList` — that we loop through each item in the list, using a `repeat...while` loop. If you click a hot spot, it gives an alert and the rect data.

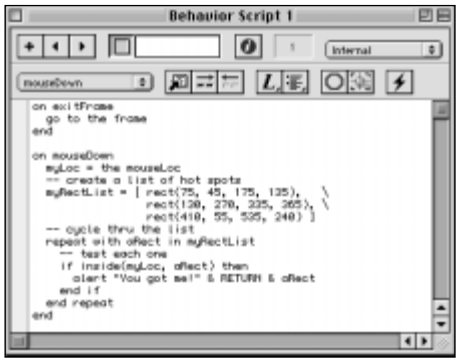

Figure 12-23: Testing for various hot spots on the Stage

intersect()

This function returns a rect, and the dimensions of the rect are the size of the overlap between two rects. If the two rects do not intersect, then `rect(0, 0, 0, 0)` is returned.

Syntax:

```
intersect(rect1, rect2)
rect1.intersect(rect2)
```

In Figure 12-24, the gray squares represent `rect1` and `rect2`. The black square represents the return value (a rect). Imagine if the two gray squares were not touching, there would be no intersecting rect to return. That explains the return value `rect(0, 0, 0, 0)` — a rect of no dimension.

Create some rects in the Message window by typing the following code:

```
rect1 = rect(50, 50, 100, 100)
rect2 = rect(75, 75, 200, 200)
```

Test whether they intersect, as follows:

```
put intersect(rect1, rect2)
-- rect(75, 75, 100, 100)
```

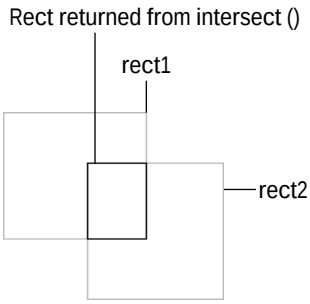

Figure 12-24: The result of two rects intersecting

You can do this with dot notation if you prefer:

```
put rect1.intersect(rect2)
-- rect(75, 75, 100, 100)
```

Now create a third rect that does not overlap `rect2`:

```
rect3 = rect(200, 200, 300, 300)
put intersect(rect2, rect3)
-- rect(0, 0, 0, 0)
```

map()

This function takes a point or rect in one rect (the second argument) and returns the point or rect relative to the other (the third argument).

Syntax:

```
map(point1, rect1, rect2)
map(rect1, rect2, rect3)
```

If you prefer not to follow the steps, refer to map.dir in the EXERCISE:CH12 (EXERCISE\CH12) folder on the accompanying CD-ROM.

This example requires a few steps, so follow along:

1. Using the rectangle shapes from the Tool palette, create two rectangles on the Stage.
2. Create a bitmap dot and put it in sprite 3. We're using a bitmap so that we can center the regPoint of the member. Your screen should look like Figure 12-25.
3. Create the frame behavior shown in Figure 12-26.

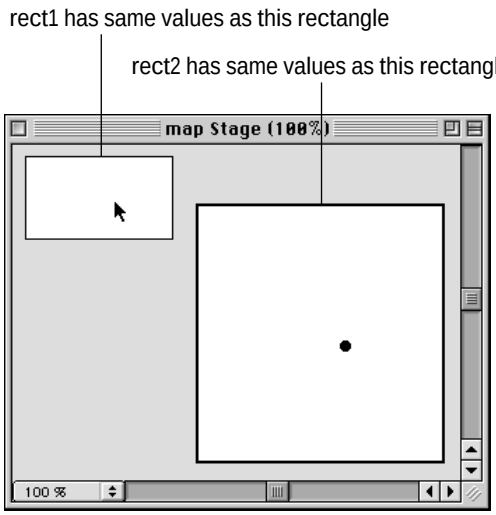

Figure 12-25: Layout of the Stage

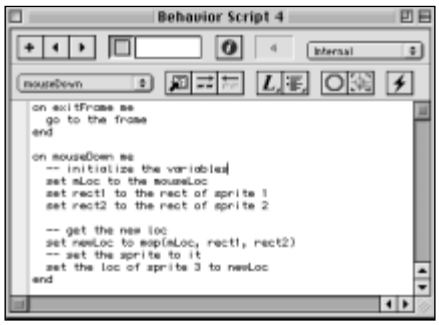

Figure 12-26: Frame behavior using the map() function when the mouse is clicked

4. Click the Recompile All Scripts button.
5. Play the movie (Control➪Play).
6. Now click around in the small rectangle. Note how the sprite moves around in the relative location in the larger rectangle.

The CD-ROM that accompanies this book contains a file called map2.dir in the EXERCISE:CH12 (EXERCISE\CH12) folder that illustrates an alternative approach, in which we specify rects in the code, using an image of two boxes.

This exercise illustrates another aspect of the `map()` function: The two rectangles do *not* have to be similar in proportion. As you might have noticed in Figure 12-27, we simply picked up the rects of the sprites.

offset()

The offset function enables you to change a rect's position. It returns a rect that has its left and right, or its top and bottom, or both, properties modified.

Syntax:

```
offset(myRect, horz, vert)
myRect.offset(horz, vert)
```

In Figure 12-27, the gray square represents the rect passed into the function, and the black square represents the rect returned by the function. In this case, the rect is being shifted more to the right than down. If you wanted to move the invisible hot spot created in Figure 12-22, `offset()` is the function to use:

```
myRect = rect(0, 0, 100, 100)
put offset(myRect, 20, 50)
-- rect(20, 50, 120, 150)
```

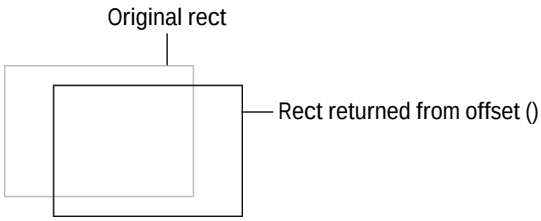

Figure 12-27: Moving a rect

union()

This function returns a rect that encompasses both of the rects passed in.

Syntax:

```
union(rect1, rect2)
rect1.union(rect2)
```

In Figure 12-28, the gray squares represent `rect1` and `rect2`, and the black square represents the return value (a rect). Note that the rect returned is the smallest possible rect that can enclose the two:

```
set rect1 to  rect(57, 50, 84, 77)
set rect2 to rect(62, 127, 117, 176)
put union(rect1, rect2)
-- rect(57, 50, 117, 176)
```

The `union()` function does not require that the two rects touch.

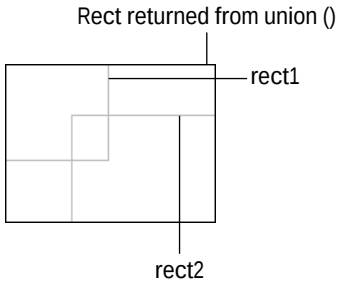

Figure 12-28: The union of two rects

Converting between rects, points, and lists

You can do arithmetic operations between rects, points, and lists. The first operand dictates the type of the resulting value. If you add a rect and list, you get a point. If you add a list and a rect, you get a rect. The following code illustrates this in the Message window:

```
put rect(1, 2, 3, 4)  +  [1, 2, 3, 4]
-- rect(2, 4, 6, 8)
put  [1, 2, 3, 4] + rect(1, 2, 3, 4)
-- [2, 4, 6, 8]
put point(1, 2) + [3, 4]
-- point(4, 6)
put [3, 4] +  point(1, 2)
-- [4, 6]
```

The data type you use first is the type you are returned. This is not the case when you add rects and points:

```
put rect(1, 2, 3, 4)  +  point(1, 2)
-- [2, 4]
put point(1, 2)  +  rect(1, 2, 3, 4)
-- [2, 4]
```

The result is a list whenever you add a rect and a point. This also occurs when you add rects or points to lists that have a different number of elements than the rect or point, as follows:

```
put rect(1, 2, 3, 4)  +  [1,2]
-- [2, 4]
put [1, 2]  +  rect(1, 2, 3, 4)
-- [2, 4]
```

Colors

The rgb and paletteIndex color data structures are new types in Director. Where the `ilk` function identifies rects and points as lists, it does not see rgb colors as lists. Nevertheless, you can treat them like lists in many ways. You can't use any of the `get` and `set` functions, but you can do arithmetic operations on them, and you can access the red, green, blue properties in a way similar to accessing to property lists.

Syntax:

```
color(#rgb, r, g, b)
color(#paletteIndex, index)

rgb(hexStr)
rgb(r, g, b)

paletteIndex(index)
```

When you create a new color with the color function, the value returned is of the type `#rgb` or `#paletteIndex`. It is in the form `rgb(r, g, b)` or `paletteIndex(index)`, respectively. Now do a little color experimentation:

```
set newColor to color(#rgb, 204, 55, 20)
put newColor
-- rgb( 204, 55, 20 )
put newColor.red
-- 204
put the blue of newColor
-- 20
put newColor.colorType
-- #rgb
put newColor.hexString()
-- "#CC3714"
put newColor.paletteIndex
-- 65

set palColor to color(#paletteIndex, 35)
put palColor
-- paletteIndex( 35 )
```

When you get the paletteIndex of an rgb color, the index of the closest color on the current palette is returned. In this case, we are using the System-Mac color palette. You can force an rgb color-type to the paletteIndex color-type. Watch out, though, when you convert to a palette and then back to an rgb, because you most likely

won't get the same color back. Continuing with the preceding example, note how `newColor` — when converted to `#paletteIndex` and then back to `#rgb` — has had its green and blue values change:

```
newColor.colorType = #paletteIndex
put newColor
-- paletteIndex( 65 )
newColor.colorType = #rgb
put newColor
-- rgb( 204, 51, 0 )
```

If you start with the paletteIndex, convert to rgb, and then convert back, you won't lose any information, because the palette colors are a subset of the rgb colors:

```
newColor = paletteIndex(35)
newColor.colorType = #rgb
put newColor
-- rgb( 255, 0, 0 )
newColor.colorType = #paletteIndex
put newColor
-- paletteIndex( 35 )
```

The results of arithmetic operations are from 0 to 255. The result does not exceed 255, nor is it less than 0. Additionally, values are always integers (note how 55, the green value in the following example, is evaluated to 27 after it is multiplied by .5).

```
set newColor to color(#rgb, 204, 55, 20)
put newColor * .5
-- rgb( 102, 27, 10 )
put newColor * 100
-- rgb( 255, 255, 255 )
put newColor * -1
-- rgb( 0, 0, 0 )
```

You can use these values to change the color of sprites. It works best on sprites whose member is a 1-bit bitmap. It also works well on fields and text cast members, as well as on shape or vector members.

Dates

Lingo now sports a date format. Like colors, the date data type is not identified as a list by the `ilk` function. Nevertheless, it, too, shares many list-like qualities.

Syntax:

```
date(string)
date(integer)
date(year, month, day)
```

The parameters for the date format must be in the following format:

```
put date("20000601")
-- date( 2000, 6, 1 )
put date(20000601)
-- date( 2000, 6, 1 )
```

Regardless of which format you use, a date structure is returned. But you must always put it in the year, month, day format:

```
set myDate to date(2000, 6, 1)
put the year of myDate
-- 2000
put myDate.month
-- 6
```

One of the neat things that you can do with dates is to add or subtract them:

```
set xmas to date(2000, 12, 25)
set myDate to date(2000, 6, 1)
put xmas - myDate && "of days until Christmas!"
-- "207 of days until Christmas!"
```

Another interesting feature of dates is when you `put` a wrong one. For example, June only has 30 days. Look what happens when we enter 31:

```
put date(2000, 6, 31)
-- date( 2000, 7, 1 )
```

It rolled over to reflect an actual date. It works for months as well:

```
put date(2000, 13, 1)
-- date( 2001, 1, 1 )
```

Summary

Before you continue and create parent scripts and child objects in Chapter 13, recall some of the things that this chapter revealed about lists:

- ✦ Lists are collections of data that can be stored in a single variable.
- ✦ There are two kinds of lists: linear and property.
- ✦ You access linear lists by position. You access property lists by position or property name.
- ✦ There are many ways to get and set list values.

- ✦ Processing items in lists is much faster than processing items in strings.
- ✦ You can manipulate rects and points like lists, and they have many functions of their own.
- ✦ Colors and dates are complex structures that have some similarity to lists.

Chapter 13 discusses using Lingo for object-oriented programming.

✦ ✦ ✦