{ Chapter 8 }

Functions

Now that you can write complex statements and affect clip instances by changing their properties, it's time to learn to modularize your scripts. Placing the same (or very similar) script on several buttons is a cry for help. Every time you copy and paste the same block of code, a little voice should be saying "No!" In Chapter 3, "The Programmer's Approach," we discussed *why* you want to reduce repeated code. Now you'll see one great way to do it: functions.

Among all the benefits of functions, the core benefit is that you can store code in one place and access it as much as you want. Type it once, use it a million times. This chapter will introduce other benefits as well as show you how to create functions.

In addition to learning how to use homemade functions just as you use Flash's built-in functions, in this chapter you will write functions that do the following:

- Act as subroutines, thus eliminating repeated code
- Accept parameters so that they can perform differently based on different situations
- Return values so that they can be used within expressions
- Act as custom methods (for your own purposes)

You'll see all three uses for functions (subroutines, returning values, and acting like methods). As you might have noticed, functions can do more than simply reduce repeated code—but that's the main thing. Regardless of how you use them, functions always take the same form.

How to Use Functions

Functions involve two steps: writing the function and then using the function. We're going to discuss using functions first, which might seem like I'm putting the cart before the horse. However, because Flash's built-in functions are already written, it's easy to look at using those. Also, this way, when you do write your own functions, you'll already know how to use them. Suffice it to say that you can't start using a homemade function unless you write it first…and if you just write a function (but never use it), nothing happens.

Using Built-in Functions

To get this far in the book, you've already learned *something* about the built-in functions. In addition to the brief explanation in Chapter 4, "Basic Programming in Flash," you actually used a few functions in Chapter 5, "Programming Structures." For example, to ensure that an expression was treated like the number data type, we used the Number() function as in `Number(anExpression)`. This evaluates the string "anExpression" as a number. If the expression in parentheses evaluated as `"112"`, the entire expression (`Number("112")`) would evaluate as `112`.

In Chapter 5, we also looked at the methods of the Math object (square root and sine, for example). These are—at their core—functions too, but we'll return to methods later in this chapter. For now, let's consider only the conventional functions listed in the Actions panel (as in Figure 8.1).

Regarding all the built-in functions, consider two things: how they are used in expressions or statements and what they do. Any time you want to call (that is, execute) a function you simply type its name and parentheses using this form: `functionName()`. Functions' names are always followed by parentheses. Not only does this make them easy to identify, but—more importantly—the parentheses provide a means for you to provide an optional parameter (also sometimes called an *argument*). The `getTimer()` function is complete without parameters. `getTimer()` returns the elapsed milliseconds since the movie started, so it

doesn't need any additional information. The `Number()` function, however, requires a parameter. It needs to know what expression to evaluate as a number. Another function that requires a parameter, `String(`*`expression`*`)`, will return a string version of *`expression`*.

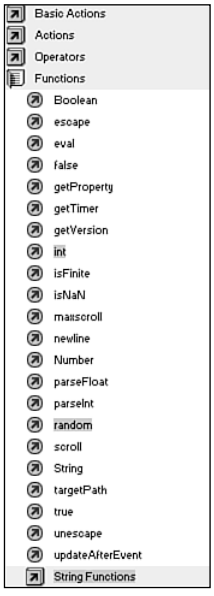

Figure 8.1 *Flash's conventional functions as listed in the Actions panel.*

Now you know how to use a function: call its name followed by parentheses, which may or may not accept parameters. As for what they do, it's important to understand that the built-in functions do *nothing* except return values. They don't perform an assignment as a statement does. That is, if `myVar` equals the string `"11"`, the expression `Number(myVar)` not only has no effect on `myVar` (it'll stay `"11"`), but by itself `Number(myVar)` is practically meaningless because it's only an expression and not a statement. The function only *returns* a value so you can use the function within a larger expression or statement. For example, `myVar= Number(myVar)` will first perform the function on the right side of the equal sign and return (into its place) a number version of `myVar` (so the statement becomes `myVar=11`). Then the equals will perform the assignment (and `myVar` is changed). It's simply a way to write an expression that changes depending on what the function returns.

Imagine a function that returned the effective temperature based on the wind chill. (By the way, we could write such a homemade function.) In that case, you'd need two parameters (current temperature and wind speed). Multiple parameters are separated by commas. Calling the homemade function would look like this: `effectiveTemp(40,20)`. We could build the function so that it arbitrarily establishes the first parameter is current temperature and the second is wind speed. Even after we build such a function, we can't just call the function because we need a place for the answer to go—that is, by itself `effectiveTemp(40,20)` doesn't really *do* anything. One logical thing you can do is call this function within a larger expression, which would look like this: `"It's only 40 degrees, but it feels like it's "+effectiveTemp(40,20)+"!"`. You could also call the function within a statement to assign the result that's returned to a variable: `realTemp=effectiveTemp(40,20)`. The thing to remember is that all built-in functions do one thing: return values.

Another point that should be clear is that the parameters you provide can be hard-wired (such as `40` or `20`), or they can be variables (such as `curTemp` or `curSpeed`) and the value of the variables will be used instead. The parameters could actually be the result of expressions (such as `TempSum/NumSamples` or `speedInKilometersPerHour*.62`). Whatever is placed in the parentheses will be evaluated. Finally, consider that expressions can include calls to functions that return values. Therefore, one parameter might invoke a function and whatever is returned from that function would be used in its place. These examples are completely legitimate: `realTemp=effectiveTemp(curTemp,speedInKilometersPerHour*.62)` and `realTemp=effectiveTemp(Number(temperatureString), Number(windSpeedString))`. Notice the `Number()` function that's nested in place of a parameter. Just remember all that you learned in Chapter 5 about writing complex expressions and the nested parentheses should be easy to track.

Using Homemade Functions

Using homemade functions is practically identical to using built-in functions. You call homemade functions in the same way that you call built-in functions: `functionName(optionalParams)`. The difference is that homemade functions can do other things besides *just* return values. You can design your homemade functions to return values if you want. Also, if you want your function to accept parameters, you need to build them that way. So, calling homemade functions is

identical to calling built-in functions. However, when we get to writing our functions (later this chapter), you'll find that homemade functions can do a lot more than the built-in ones—you just have to write the script to make them perform the operation that you have in mind.

There is one slight difference in how you call homemade functions. In the case of the built-in functions, you can call them any time, any place—on a button, in a clip's keyframe, wherever. Homemade functions are written in keyframes. It can be a keyframe in the main timeline or inside a nested clip. Because homemade functions are "in" a particular timeline, they need to be targeted. If you are calling the function from any keyframe or button located in the same timeline as your function, you can call that function by simply typing its name (`functionName()`). (That is, technically, a relative reference.) However, if you are "in" another timeline, you have to precede the name of the function with a target path. Maybe you have a function in the main timeline and you want to call it from inside a clip. Just use the absolute reference `_root.functionName()`. If the clip is only nested one level deep, you could alternatively call the function with the relative `_parent.functionName()`. The concept of targeting should be very familiar to you. All the same information you learned about targeting properties, variables, or methods of clips in Chapter 7, "The Movie Clip Object," applies to calling functions…relative and absolute references. (You can learn more about targeting in Chapters 1, "Flash Basics," and 7.)

In Chapter 7, you also learned that many built-in Actions are really methods of the Movie Clip object. Methods are functions that are applied to individual instances (of clips, in this case). For instance, as a method, `someClip.gotoAndStop(2)` will cause an instance named "someClip" to jump to frame 2. When you write homemade functions, you can choose to write them in a keyframe of the main timeline or in a keyframe of any master symbol. Naturally, you'll need an instance of the clip containing the function if you want to call it. When calling such functions, you always precede the function name with a path to that function. The syntax of such a call looks the same as when applying methods to clips. That is, *`someClip`*`.gotoAndStop(2)` is the same form as *`someClip.myFunction()`* (where "myFunction" is the name of a homemade function that exists in a keyframe of the clip "someClip"). Not only do they "look" the same, but homemade functions can act like built-in methods if that's how you design them. This is a great way to leverage the knowledge you already have. Instead of trying to learn lots of different things, I think it's best to learn a few things really well, and then everything else can be understood in relative terms.

Creating Homemade Functions

Now that you know how to call functions (that is, how to use them), we can look at how you write them. The concepts of accepting parameters, returning values, and acting like methods should start to really make sense when you apply them to a purpose. You'll not only learn how to write functions, but also how they can help you. It's not as though this were an exercise in learning vocabulary words such as *parameters* and *methods*; rather, you will get to the point where you can reach for these tools as needed to solve problems.

Basics

Functions are written in keyframes. I find it much easier to type in the basic form while in Expert Mode. Here's the skeleton form:

```
function myFunction(){
}
```

The word "function" is *always* used as is. Next, you type the name for your function—anything you want as long as it's one word. (Keep in mind, you can't use words that are already part of the ActionScript language for your function name.) Parentheses always follow the function name. If you expect to receive any parameters, you must provide each with a temporary one-word name (separated by commas when you have more than one parameter). Finally, the opening and closing curly brackets enclose your entire script. Within the script for the function, you can use the name given for any parameters and it will evaluate as the value for that parameter. For example, consider this start of a function that accepts a parameter:

```
function doubleIt(whatNum){
}
```

In this case, the function name is "doubleIt." If you call this function (from elsewhere), you'd say `doubleIt()`. Because this function can be called while providing a parameter, the call would actually look like `doubleIt(12)` or like `doubleIt(getTimer())`. Calling the function effectively jumps to the function, sending with it the value of the parameter. Once at the function's script, the value provided as a parameter is referred to by using the parameter name (in this case, `whatNum`). If the parameter's value happens to be `12`, `whatNum` is `12`; if the parameter is `1203`, `whatNum` is `1203`. This is just like any variable (you refer to their

values by referring to their name). In the preceding example, you can refer to `whatNum` anywhere within the function, and you'll be referring to the value of whatever was passed as a parameter. You'll see how to apply parameters in a moment, but for now just understand these two basic forms (functions that accept parameters and those that don't).

You're about to see the four basic applications of homemade functions. Then you'll have a chance to create functions that solve problems.

Functions as Subroutines

The first type of function we're going to write is unlike built-in functions because it won't return a value. A *subroutine* is one or more lines of code that you want to execute from more than one place in your movie. Perhaps you have several buttons that do the same basic thing. Instead of putting the same script on each button, you can call the same function from each button. The advantage (in addition to reducing the amount of typing) is that your code is centralized. If there's a bug or you want to make an adjustment, you need to do it in only one place (in the master function) and not on each button's call to the function.

Consider the example from the "Dynamic Referencing" section in Chapter 7 (shown in Figure 8.2).

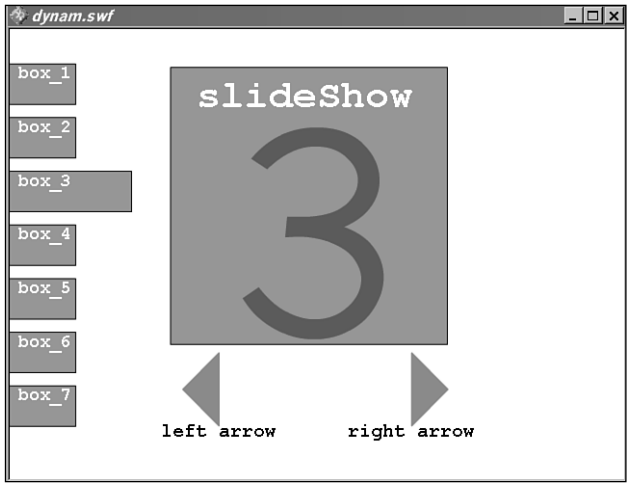

Figure 8.2 *Moving all seven "box_" clips can be done in a for-loop that references them dynamically.*

There were "forward" and "back" buttons that, in addition to moving to the next or previous frame of a clip, also set the _x position of seven clips ("box_1," "box_2," and so on) to zero. The code for the "forward" button was as follows:

```
on (release) {
    slideShow.nextFrame(); //move slide show ahead
    //Move all the boxes back to 0
    for(i=1;i<8;i++){
        _root["box_"+i]._x=0;
    }
    _root["box_"+slideShow._currentFrame]._x=50; //set cur box to 50
}
```

Notice that the "back" button code was almost identical:

```
on (release) {
    slideShow.prevFrame(); //move slide show back
    //Move all the boxes back to 0
    for(i=1;i<8;i++){
        _root["box_"+i]._x=0;
    }
    _root["box_"+slideShow._currentFrame]._x=50; //set cur box to 50
}
```

In addition to the "back" and "forward" buttons, there were seven invisible buttons on top of the box clips that had code that was nearly identical. Each of the seven buttons covering the box clips had code like this (although the parameter for `gotoAndStop()` was different for each button):

```
on (release) {
    slideShow.gotoAndStop(1); //move slide show to frame 1
    //Move all the boxes back to 0
    for(i=1;i<8;i++){
        _root["box_"+i]._x=0;
    }
    _root["box_"+slideShow._currentFrame]._x=50; //set cur box to 50
}
```

The only difference in each of the seven buttons was that the frame number used in the `gotoAndStop()` method was different: 1, 2, 3, and so on. But the rest of this code is the same as the "forward" and "back" buttons.

This is clearly a case in which a function can serve to eliminate redundant code. The script in each button is identical except for the first line (and, as you'll see later, the first line is even similar enough to be moved into a function that accepts

a parameter). For now, let's move the repeated code from each button into a function. In place of the code that's moved, we simply call our function. So, you can simply create a function in frame 1 of the main timeline:

```
function moveBoxes(){
}
```

Then, paste (between the curly brackets) the code taken from each button. The finished function will look like this:

```
function moveBoxes () {
     for(i=1;i<8;i++){
          _root["box_"+i]._x=0;
     }
     _root["box_"+slideShow._currentFrame]._x=50; //set cur box to 50
}
```

Finally, you simply need to call this function from each button. The "next" button becomes

```
on (release) {
     slideShow.nextFrame(); //move slide show ahead
     moveBoxes();
}
```

The "back" button becomes

```
on (release) {
     slideShow.prevFrame(); //move slide show back
     moveBoxes();
}
```

And each invisible button looks like this:

```
on (release) {
     slideShow.gotoAndStop(1); //the 1 is different in each button
     moveBoxes();
}
```

Notice that in place of the code that was moved to the function, a call to the function is used instead. That is, `moveBoxes()` is used in place of the code that was removed.

Because the code is in only one place, it can be modified quickly. For example, if it turns out that there are more than seven "box" clips to move, we can modify that for-loop in the function.

The process of writing a function is to first identify a need and then solve it. In the case of a function that serves as a subroutine, the need is to reduce redundant code. The solution involves extracting that portion of the code that's repeated, moving it into a function, and—in the place from which it was extracted—calling the function. It's fine to start scripting and later notice that some code is repeated. When you find yourself copying and pasting code, bells should ring in your head saying, "time to consider a function." I often build my first version of a script using a rather hard-wired approach. After I get it working, I walk through the code and try to identify portions that are duplicated. Then, I try to move the duplicated code into a function instead.

As you're about to see, the repeated code doesn't even have to be identical. It can simply be similar. The bells that ring in your head can also be useful if they identify portions of your code that follow the same pattern. Just think about pseudo-code. If the explanation of what's being achieved in your code (the pseudo-code) can be generalized, you can probably write a function instead. For instance, in the preceding example, we didn't extract the very first line in each button because they were different. One used `nextFrame()`, another `prevFrame()`, and each of the seven invisible buttons used a different parameter for `gotoAndStop()`. Although this might seem unique for each button, it can actually be generalized. In pseudo-code, the general version of the first line for *each* button is "jump slideShow to a new frame." The trick is translating the pseudo-code. You'll see that the solution is to use a parameter.

Making Functions That Accept Parameters

Writing a function that accepts parameters is quite easy. Doing it effectively is just a bit more work. First, consider the form

```
function myFunction(param){
}
```

Whatever value is sent as a parameter when calling this function (as in `myFunction(12)`) can be referred to by using the variable name `param`. Inside the function (between the curly brackets), you can refer to that parameter name (`param`, in this case) and you are really referring to the value sent from the function call. It's like if you order a steak cooked "well done." Consider that the cook always performs the "cookIt" function. The parameter is "doneness." It doesn't matter whether you call this function by saying cookIt("wellDone") or cookIt("rare"), there's always a "doneness" parameter. It just happens that the value for doneness varies.

One common reason to make your function accept parameters is that you don't really want to perform the *exact* same procedure every time, but rather you want to perform a slightly different procedure each time. Just like the "cookIt" function, you'd like some variation available. Let's try to further consolidate the script in each button from the last example. The "next" button uses `slideShow.nextFrame()`, the "previous" button uses `slideShow.prevFrame()`, and the seven other buttons use `slideShow.gotoAndStop(x)` (where "x" is 1 through 7). Although this might look like three distinct scripts, they can easily be consolidated. Without changing what we've already coded in `moveBoxes`, we can add a feature to this function. Namely, we can make it accept a parameter that serves as the destination frame for the `slideShow` clip. That is, `slideShow.gotoAndStop(`*`destinationFrame`*`)` will work great if "destinationFrame" evaluates to the correct number. We'll just send a number when we call the `moveBoxes()` function (as in `moveBoxes(2)`) and name the parameter `destinationFrame`. Check out the finished function:

```
function moveBoxes (destinationFrame) {
     slideShow.gotoAndStop(destinationFrame);
     for(i=1;i<8;i++){
          _root["box_"+i]._x=0;
     }
     _root["box_"+slideShow._currentFrame]._x=50; //set cur box to 50
}
```

Notice that only the very first line and the second line have changed (the rest remains untouched). Now that this function accepts parameters (namely, the frame to which you want `slideShow` to jump), we can adjust the various calls to this function. (By the way, only when you make a significant change to the function—like adding a parameter—do you need to modify every call to that function—usually edits will occur only in the function itself and not in the calls to the function.) The seven buttons are easy to adjust. In each button, remove the line that starts `slideShow.gotoAndStop()` and change `moveBoxes()` to `moveBoxes(1)` for the first button, `moveBoxes(2)` for the second button, and so on. For the "forward" and "back" buttons, you need to first remove the first line (either `slideShow.nextFrame()` or `slideShow.prevFrame()`). Then when calling `moveBoxes()`, you need a value for the parameter. You can't just hard-wire something like `moveBoxes(2)` because that will *always* jump to frame 2. The "forward" button should (in pseudo-code) "jump to the current frame plus one" and the "back" button should "jump to the current frame minus one." We can write an expression in place of the parameter that results in the frame to which we want to

jump. The call from the "forward" button will look like this: `moveBoxes (slideShow._currentFrame+1)`. The "back" button will use `moveBoxes (slideShow._currentFrame-1)`. The expression `slideShow._currentFrame+1` can be translated as "slideShow's current frame plus one."

Finally, there's one slight problem with the solution I've outlined. Namely, it's possible to press the "forward" button when you're already on the last frame of "slideShow" or press the "back" button when you're on the first frame. Therefore, the value that is sent as a parameter can be too high or too low. Inside the function, the line `slideShow.gotoAndStop(destinationFrame)` will attempt to jump to frame zero or to a frame number greater than the maximum. Nothing detrimental happens, but it's worth addressing this issue…for practice, if nothing else. (Ideally, we'd just make the buttons dim out and become inactive appropriately—and you'll do just that in the Slide Show workshop.) Without going through the work to inactivate buttons there's another simple fix for this issue. Inside and at the top of the `moveBoxes()` function, add the following two if-statements:

```
if(destinationFrame==0){
  destinationFrame=1;
}
if(destinationFrame>slideShow._totalFrames){
  destinationFrame=slideShow._totalFrames;
}
```

Translated, the first if-statement says that if the value for `destinationFrame` happens to be `0`, reset `destinationFrame` to equal `1`. The second if-statement checks whether `destinationFrame` is greater than the `_totalFrames` property of `slideShow` and if so, it sets `destinationFrame` to equal `_totalFrames`.

Just because `destinationFrame` is a parameter that's accepted doesn't prevent us from changing its value after we're inside the function. This solution resolves the minor flaw in the original function. Here's the final function in case you want to attempt to rebuild the example from chapter 7:

```
function moveBoxes (destinationFrame) {
  if(destinationFrame==0){
    destinationFrame=1;
  }
  if(destinationFrame>slideShow._totalFrames){
    destinationFrame=slideShow._totalFrames;
  }
```

```
    slideShow.gotoAndStop(destinationFrame);
    for(i=1;i<8;i++){
        _root["box_"+i]._x=0;
    }
    _root["box_"+slideShow._currentFrame]._x=50; //set cur box to 50
}
```

It's both typical and desirable to put the bulk of your code in functions and then make the calls to that function as minimal as possible. Remember, you can invoke any function as many times as you make calls to it.

Even though this sample function accepted a parameter and used that parameter's value directly (as the frame to which we jumped), parameters don't have to be used so directly. The parameter can control what part of a function to skip or execute. For example, a function could perform several very different procedures depending on the parameter accepted. Consider this example:

```
function doSomething(whatToDo){
  if (whatToDo=="eat"){
    //place code for "eating" here
  }
  if (whatToDo=="sleep"){
    //place code for "sleeping" here
  }
}
```

If the function is called with `doSomething("eat")`, just the code within the first if-statement is executed. Notice, too, that if you called `doSomething("cry")`, neither if-statement will be entered. Of course, you can also write nested if-else or if-else-if statements. The point I'm making here is that you can use the parameter to affect which part of the function is executed, rather than using the parameter's value directly within an assignment inside the function. I use this technique often for multipurpose functions, which act like a clearing house. Several different procedures go through the same function, but only execute a small portion of the function.

Making Functions That Return Values

Making a function that returns a value is as simple as adding a line that starts with `return`. Following the word `return`, you can type a hard-wired number, a variable, or an expression—the value of which will be "returned" to wherever the function was called. Consider this basic form:

```
function doubleIt(whatNum){
  return whatNum*2;
}
```

Now, from anywhere in your movie, you can call this function. Because this function returns a value, the place where you call the function turns into the value that's returned. So, `trace(doubleIt(12))` will display `24` in the output window. You could also say this:

```
theAnswer=doubleIt(22);
trace("Two times 22 is "+theAnswer);
```

One important note about the word `return`. In addition to specifying what is returned (to wherever the function is called), this will jump out of the function. That is, if there are more lines of code after `return` is encountered, they'll be skipped. This is actually kind of nice even if you're not trying to write a function that returns a value. For example, an if-statement at the top of a function could cause the rest of the function to be skipped when a particular condition is met. We looked at this technique in Chapter 5 and compared it to `break`—which only jumps out of an enclosed loop (not the entire function the way `return` does).

The main thing to remember about functions that return values is that you'll probably want to call them from within a statement. Simply writing the script `doubleIt(12)` doesn't really do anything because the answer (the value `24` that is returned) is not being used anywhere. There's no rule that says you have to use what's returned from a function. It's just more likely that when you call a function that returns a value, you will want to use that value somehow. Compare it to using a slot machine (you "call" the slot machine function by pulling the arm). Normally, you would take the winnings that are "returned," but if you want, you can just watch the pretty shapes spinning.

Let's look at a more practical example than my `doubleIt()` function. We can write a simple function that uses a currency exchange rate to calculate the value in U.S. dollars for a price given in Canadian dollars. The idea is that anytime you're given a price in Canadian dollars, you can call the `convert()` function (with the value in Canadian dollars as a parameter) and the value in U.S. dollars will be returned into the place the function is called. For example, you can call this function like so:

```
Trace("20 dollars Canadian is really "+convert(20)+" in US dollars");
```

This function is explored in great detail in the "Currency Exchange Calculator" workshop, but here's a finished version:

```
function convert(amountInCAD){
  exchangeRate= 0.62;
  return amountInCAD*exchangeRate;
}
```

The only reason I use the variable `exchangeRate` is that I want a clear and easy way to adjust that value (because it obviously varies). You could consolidate this into one line if you simply used `0.62` in place of `exchangeRate` in the second line. Actually, you could also add some fancy features that rounded off the answer. When you see the "Currency Exchange Calculator" workshop, you will see all kinds of fancy features—such as making the answer appear in "money format" ($1.50, not 1.5, for example). The methods of the Math object explored in Chapter 5 (as well as the String object that you'll see next chapter) will make this process relatively simple. As with all functions, those that return values aren't particularly difficult to write. The effort comes in designing a good one. You'll build your skills with practice.

Finally, it's not necessary that a function that returns a value must also accept parameters. It just makes sense when you want the function to do something *with* a value you provide.

Using Functions as Methods

Built-in functions can be called from anywhere by simply referring to the function name (as in `Number(anExpression)`). Unlike built-in functions, for home-made functions, you have to target the timeline where the function exists. Often, I write all my general purpose functions in the main timeline. If I want to call such a function from within a clip or nested clip, I have to remember to include `_root.` before the function's name (as in `_root.convert(12)`). As previously mentioned, a function that's written in a keyframe of a different timeline needs to be targeted as well. You could actually have two different Movie Clips each with a function named `myFunction()` in their first keyframe. These functions could produce entirely different results. Within either clip, simply calling the function (as `myFunction()`) would work great. If you were outside the clip or wanted to target the `myFunction()` of another clip, you'd have to precede the name with a path. For example, `_root.someClip.myFunction()` would execute the `myFunction` inside the clip with an instance name of "someClip".

To understand creating functions that perform like methods, recall what a method is. A method is a function that is applied to a single instance of a movie clip. (Actually, methods are functions that affect objects—but the object with which we're most familiar is a movie clip instance.) The "Action" `gotoAndStop(1)` is really a method because it is applied to the timeline in only one clip at a time. If you design them right, custom functions can act just like methods.

Let's write a function that serves as a method. I'd like a method called `grow()` that will increase both the `_xscale` and the `_yscale` properties of a clip (it's always such a pain to set *both* these because there's no "`_scale`" property). First, make a clip by drawing a circle, selecting it, and choosing Convert to Symbol. Then go inside the master clip and attach this script to the first frame:

```
function grow(){
  _xscale+=10;
  _yscale+=10;
}
```

Translated, this says, set the `_xscale` to `10` more and set the `_yscale` to `10` more. Which `_xscale`? Because no clip is targeted, the clip itself will grow. Now, this function can be called from anywhere inside the clip simply by saying `grow()`, but I want to do it from the main timeline. Drag a few instances of this clip to the main timeline, and then name each instance something unique (maybe `circle_1`, `circle_2`, and so on). Now, in the main timeline, create two buttons, one with this script:

```
on (release) {
  circle_1.grow();
}
```

The other button's script can be

```
on (release) {
  circle_2.grow();
}
```

Check out Figure 8.3.

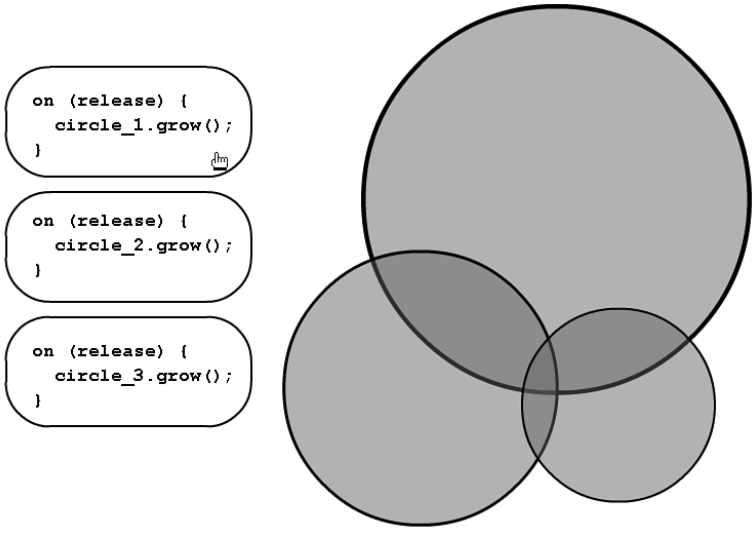

Figure 8.3 *A function inside the master symbol acts like a method of each instance.*

It looks exactly like applying a method to a clip (like you might do `circle_1. nextFrame()`).This example really does behave like a method for one important reason. The function refers (relatively) to the clip in which it is contained. I don't think this is a hard-and-fast rule of what makes a custom method, but for a function to act like a method, I think it's fair to say that the function has to affect the clip it's inside. All methods are functions—not all functions are methods. When functions are unique to the clip in which they're contained, you can think of them as methods.

Local Variables

The variations of functions (acting like methods, returning values, accepting parameters, and acting like subroutines) are all part of the same thing: functions. They're not even exclusive concepts. For example, you can have a subroutine that accepts parameters. The differences are in the way you use the functions you create. Local variables are another concept related to functions. You can use local variables in any type of function, but you don't have to.

Local variables are used just like any other variable except they exist only while inside the function. Similar to the way a named parameter has a value only while you're inside the function, local variables can be accessed only from within the function. The only real benefit of local variables is that they cease to occupy any

memory after they're used. This concept of "good housekeeping" is not terribly important until your movies become very complex—and even then, it's likely that the user's computer memory (RAM) is large enough to make the issue almost nonexistent. But it's worth understanding, because there's no reason to use more memory than you have to.

Normally, after you assign a value to a variable (like `username="phillip"`), a small portion of RAM is dedicated to that variable. At any time, you can ascertain the value of `username`. Even if you're in another timeline, you can access the variable by preceding its name with the path to the variable. That variable will "live" forever—even if you reassign it to an empty string (as `username=""`, for example). If you are done with the variable, you can use the `delete` statement to remove it from memory (`delete username`). Depending on your application, you might want the variable to "live" forever. Perhaps you're tracking a user's score and you don't want to flush it from memory. Just remember that even if you stop using a variable, it's still occupying a portion of RAM (unless you delete it). Such "normal" variables can be considered global variables in that they're available at any time and from anywhere (that is, they're not "local").

All variables are safe, yet temporary, storage for data. They are temporary in that when you restart the movie, they are gone (or at least reinitialized). Some variables are used so briefly that you should consider making them local variables. A local variable does occupy RAM, but as soon as you leave the function that RAM is released and the variable ceases to exist. The way that you declare a local variable in a function is by using `var`. There are two ways; you can either say `var tempVar` (where `tempVar` will be the local variable) or `var tempVar="initial"` (where `tempVar` is the local variable and you're assigning a value from the get-go—to, in this case, the string `"initial"`). Then, from anywhere inside the enclosed function, you can refer to the variable by name (you don't need to proceed with `var`).

A perfect example of where I should have used a local variable was for the `exchangeRate` variable in this function:

```
function convert(amountInCAD){
  exchangeRate= .62;
  return amountInCAD*exchangeRate;
}
```

Because `exchangeRate` was used only once for convenience—and never again outside the function—a local variable would have been more appropriate. It would look like this:

```
function convert(amountInCAD){
    var exchangeRate= .62;
    return amountInCAD*exchangeRate;
}
```

Just that simple `var` before the first use of the variable makes it local. (Also, remember that you won't be able to access the value of a local variable from outside the function.)

Here's a great analogy to understand local variables. Just remember that variables are for storage. If you're baking a cake, you'll likely need to mix all the dry ingredients before combining them with the wet ingredients. If you use a bowl to temporarily hold the flour, salt, baking powder, and so on, the bowl can be considered a local variable. You put all the dry ingredients in one bowl, mix them, and then finally pour the whole bowlful into *another* bowl that contains your eggs, milk, vanilla, and the rest. The dish in which you bake the cake is more like a regular (global) variable. You pour the whole cake mixture into this dish, bake, and serve inside the dish. You want the baking dish to stick around for a while. This analogy is best for thinking about local variables. Often you want a place to temporarily store information (the dry ingredients or the exchange rate, for example). Then when you're done, you don't need the variable (or bowl) anymore. The truth is that if you never use a local variable, you'll probably never know the difference. It becomes an issue only when you're storing (unnecessarily) an enormous amount of data in a global variable. In any case, now you know how to declare a local variable!

Applying Functions to Previous Knowledge

Now that you've seen most of the ways built-in and homemade functions behave, it makes sense to review some previously covered concepts, which happen to apply seamlessly to functions. This section is almost a summary of functions—and that's how you should see it. Here's a chance to solidify a few concepts you've heard over and over.

Review Built-in Functions

Built-in functions all return values. Some people actually define a "function" as only something that returns values. But we've seen that homemade functions don't have this requirement—Flash's built-in functions do. If you simply remember that all built-in functions return values, you'll also remember that they are used within expressions or statements. They don't create statements by themselves.

Almost all the built-in functions follow the form `functionName(optionalParam)`. Some accept more than one parameter. Both `true` and `false` are functions in that they return true or false, respectively, but they don't use the parentheses. Use `true`, not `true()`. Finally, the two "functions" `scroll` and `maxscroll` are really "properties" of variables associated with dynamic text fields. If you have a variable (for example, `myText`) associated with a Multiline Dynamic Text field, the default `myText.scroll` is `1` (meaning that the first line appears at the top of the field). If you executed the script `myText.scroll=2`, you'd see the second line appear at the top of the field (effectively making it look like it scrolled down one line, as in Figure 8.4). Both `scroll` and `maxscroll` are definitely not functions. They look like properties in the form `variable.scroll`—but, unlike other properties, these two affect variables (not clip instances).

Finally, there is a whole set of "Actions" that act very much like homemade functions. All of Flash's Actions are either methods of clips or ActionScript statements. An example of an Action that's really a method is `nextFrame()`, which applies to a specific clip's timeline (or the current timeline when no clip is specified). The majority of the Actions, however, are really just statements. Most of these structural elements of the ActionScript language were covered in Chapter 5. Although we studied both methods and functions, realize that built-in examples of each exist within Flash.

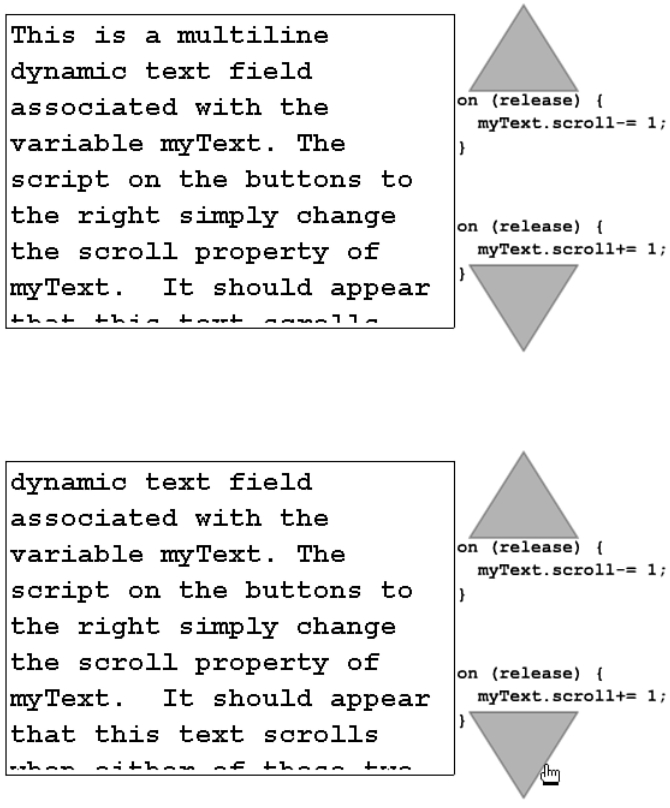

Figure 8.4 *You can make a Dynamic Text field scroll by changing the* `scroll` *property of the associated variable.*

Things to Remember

There are many things to remember when writing or calling functions. I think the biggest concept is that homemade functions are called by preceding the function name with a path to that function. Because functions are always written in keyframes, you simply need to target the timeline where it resides.

Naturally, functions that return values should be called from within an expression because the value that is returned will be returned to wherever the function was called. This concept has been explained, but realize that just because your function returns values, that doesn't mean it can't do other things, too. That is, a function can act as a subroutine (maybe setting the `_alpha` property of several clips) and when it's done, it can return a value. There's also no rule that says if a

function returns a value, you have to *use* that value. You might have a function that does several things and then returns a value. If you simply call it by name—for example, `doit()`—the value that's returned never gets used but the function still executes (including all contained scripts). Because the function returns a value, you might normally use it within a statement such as `theAnswer=doit()`, but you don't have to.

Finally, don't forget all that you learned about data types in Chapter 4. When passing values as parameters, pay attention to the data type sent to and expected by the function. Also realize functions that return values only return values of the type you specify. For example, if the following function is called using `doit("one")`, you'll have trouble because the parameter being sent is a string and the function almost certainly expects a number.

```
function doit(whatNum){
  var newLoc=whatNum*10;
  someClip._x=newLoc;
}
```

Similarly, consider the following function, which returns a string. If you call it within an expression that treats the result as a number, you'll get unexpected results.

```
function getAlpha(){
  return "The alpha is "+curAlpha
}
```

You'll also have trouble if you call the preceding function with

```
someClip._alpha=getAlpha()
```

The problem is that you're trying to set the `_alpha` of "someClip" to a string (where you can only set `_alpha` to a number). This is simply a case of mixing data types. You're trying to use apples in the orange juice maker, if you will.

Remember, too, that there's a movie clip data type. You refer clips by name but not a string version. That is, simply typing `someClip._x=100` will set the `_x` property of a clip instance called `someClip`. Notice that there are no quotation marks. The reason I'm reminding you now is that you can store a reference to a clip instance in a variable or as a parameter. For example, the following function accepts—as a parameter—a reference to a clip:

```
function moveOne(whichOne){
  whichOne._x+=10;
}
```

This function will work only when the movie clip data type is sent as a parameter. For example, if you have a clip instance named "red," you can use `moveOne(_root.red)`. If the clip is in the same timeline from which you call this function, you could use `moveOne(red)`. But notice that it's a reference to the clip (data type "movie clip"), not a string, that is being passed as the parameter.

Finally, an esoteric point should be made about the terms *argument* and *parameter*. In my opinion, they can be used interchangeably. Some people define parameter as the general term and argument as the specific term. That is, when you're not sure what the parameter's value is, it's still a parameter. After you are done analyzing and know the value, you call it an argument. So, a function can accept parameters, but when you call the function, you'll use a particular value as an argument. I'm only mentioning this definition so that you'll know *argument* and *parameter* are really the same thing. I'll try to use "parameter" throughout the book, but don't be surprised when you hear someone else say "argument."

Of course, there are countless other things to remember, but at this point, I think it makes the most sense to practice. Try to analyze a Flash movie you made in the past to see whether a function can reduce redundant code. Naturally, if it "ain't broke," there's little incentive to fix it. However, recognizing places in your own code that can be optimized is a great skill. If you're having a hard time finding flaws in your own movies, here are a few exercises to try out:

- Write a function that moves a clip instance (maybe a box) 10 pixels to the right. Create two different buttons that call this function.
- Adapt the preceding function to accept a parameter so that it can move the clip instance 10 pixels to the right or to the left—depending on the value of a parameter received. Make one button move the clip to the right, the other to the left.
- Write a different function that returns half of the value provided as a parameter. That is, if the function is called `half()`, calling the function with trace (`"Half of 4 is "+half(4)`) will result in "Half of 4 is 2" appearing in the output window.
- Write another function that acts like a method inside a clip. Make one that reduces the clip's `_alpha` or increases it. You can write two methods or one that accepts a parameter. In the end, you should be able to use buttons in the main timeline to target any particular instance (of this clip with the method) and you can reduce or increase the `_alpha`.

Summary

Functions are so useful that it's hard to imagine programming without them. It's possible (after all, you couldn't write functions in Flash 4), but functions mean that repeated code can be consolidated; that one block of code can behave slightly differently depending on the value of a parameter received; that values can be returned; and that you can create your own methods.

Throughout all these techniques, one thing remains consistent: The form of a function is always the same. Additional parameters will sometimes appear in the parentheses following the function name, and sometimes you'll return values, but the form is always the same. Just like if-statements and for-loops, you should start every function by typing the core form (always in a keyframe script) as

```
function anyName(){
}
```

Then you can fill in the space between the curly brackets and parameters if you want. Practically every workshop exercise involves a function, so get used to it! You'll learn to love the way that functions minimize typing.