

The top half of the cover features a dark, abstract background with a grid of white lines. A bright, jagged lightning bolt strikes down from the top center. The scene is composed of various geometric shapes and textures, creating a sense of depth and complexity.

CHAPTER 2

The bottom half of the cover has a dark background with a fine grid of white dots. A vertical white line runs down the center, and a horizontal white line runs across the bottom. Three small, bright white stars are scattered across the grid.

**ActionScript 2.0
Essentials**

IN this chapter, I will talk about some basic programming concepts in ActionScript 2.0. You will use your knowledge of Flash and the concepts you learn in this chapter to create games that are playable on the Wii. As you build games throughout this book, you may want to occasionally come back to this chapter to review the concepts you learned. Learning a programming language can be extremely challenging and can take a lot of time, but it will feel more and more natural to you as you work at it.

Just like the last chapter, this chapter does not contain everything there is to know about ActionScript 2.0. Rather, this chapter will teach you the basic concepts of ActionScript 2.0 and give you the skills you need to build Flash games for the Wii. At the end of this chapter, there are some resources where you can learn more about ActionScript 2.0.

UNDERSTANDING ACTIONSCRIPT 2.0

ActionScript 2.0 is the language used to program Wii Flash games. It is very similar to JavaScript, but you don't need to know JavaScript to learn ActionScript 2.0.

Where Do I Place the Code?

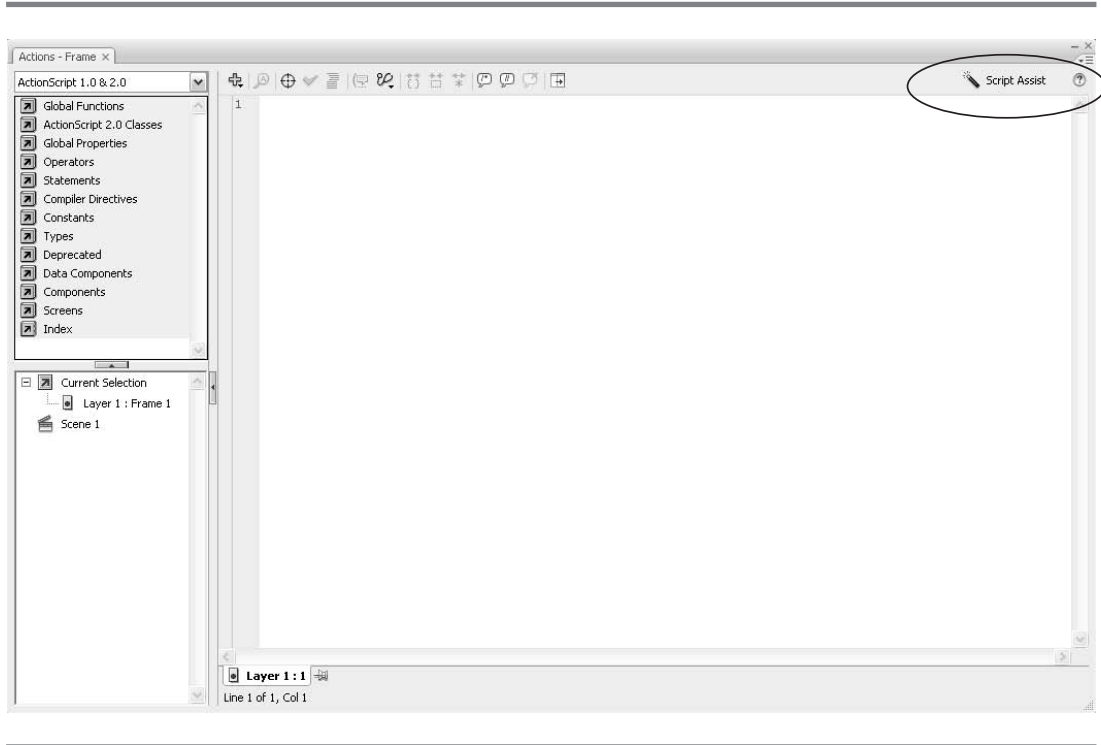
There are several places you can place ActionScript code. You can place it directly on movie clips or buttons, in keyframes in the Timeline or keyframes within movie clips, or in external files. I do not recommend placing code directly on movie clips or buttons, as this creates code that can be difficult to maintain. Placing code in external files and placing it in the Timeline are both good practices. For simplicity's sake, in this book I will place all code within keyframes in the Timeline.

How Do I Place Code in the Timeline?

Code can be placed in the Timeline using the Actions panel. The Actions panel can be accessed by choosing Window | Actions or by pressing the keyboard shortcut F9 (Windows) or OPTION-F9 (Mac). Before you open the Actions panel, always make sure you have the frame where you want to supply ActionScript selected.

There are two ways to write ActionScript in the Actions panel: Normal mode and Script Assist mode. To toggle between the two, click the Script Assist button in the Actions panel (see Figure 2-1). Normal mode requires you to type out all your code by hand, and Script Assist mode is more of an intuitive interface where Flash writes much of the code for you. Though I do not discourage the use of Script Assist, its capabilities are limited, so I will use Normal mode for the code in this book. Once you have an understanding of how ActionScript 2.0 works, and some practice typing the code, it may be faster for you to use Normal mode than Script Assist.

FIGURE 2-1



The Actions panel and Script Assist button

The area in the left side of the Actions panel is called the Toolbox. The Toolbox is for helping you write code and is used in conjunction with Script Assist. You can show or hide the Toolbox by clicking the arrow that divides the Toolbox and the Scripting pane. The Scripting pane is the main area of the Actions panel where you write code.

Making Comments in Your Code

As you write more and more ActionScript, it is always best to place comments in your code so that you or someone else can easily navigate through the ActionScript you write. Comments are not processed as ActionScript code, so you can write comments as messages to yourself or others in plain English. . .or any other language, for that matter. You can also place code in a comment to temporarily disable it. In ActionScript, comments have a gray color so that they are easily distinguishable from ActionScript code.

Making Single-Line Comments

To make a single-line comment, type two forward slashes (`//`) at the beginning of the line. This will tell Flash to skip that line of code when processing ActionScript. This is what a single-line comment looks like:

```
// This is a single-line comment
```

Making Multiline Comments

To create a multiline comment, begin the commented area with a forward slash and an asterisk (`/*`), and end the comment with an asterisk and then a forward slash (`*/`). A multiline comment looks like this:

```
/*  
Comment line 1  
Comment line 2  
*/
```

WORKING WITH VARIABLES

You are already familiar with variables if you have taken an Algebra class. In math, variables represent numbers whose value you do not know. Working with variables allows you to create equations that will work regardless of whether you know the value of the variable.

In ActionScript, *variables* are used to hold data. For example, you could create a variable called `score` that could represent the number of points a player has in a game. Storing the player's points in the `score` variable enables you to work with the player's points without having to know the exact number of points the player has.

Variables do not always contain numbers. They can contain true or false values, text, or any other type of data you can work with in Flash. In this section, you will take a look at how to create variables in Flash.

Defining Variables

To create or define a variable, type `var` in the Scripting pane in the Actions panel. After `var`, type a space and then type the name of your variable. Variable names should begin with a lowercase letter and only contain letters and numbers. To create a variable called `score`, you would type

```
var score
```

Setting Values

Variables can hold data, but in order for that to happen you must tell Flash what data the variable will hold. To set a value of a variable, use the equal sign (=) and then type the value of the variable. Placing spaces before or after the equal sign is optional, but you may find it easier to read your code if you use a space. If you wanted to give a value of 0 to a variable called score, you would type

```
var score = 0;
```

NOTE

The semicolon at the end of the code works similar to a period in a sentence; it marks the end of a statement. If you forget to put semicolons at the end of statements in your ActionScript code, you will not usually get an error and your code will run properly. However, it is a best practice to always end every ActionScript statement with a semicolon.

Sometimes, you will not give a value to a variable at the same time you create it. I'll talk about those times when examples come up, starting in the next chapter. In these cases, you can declare a variable on one line and give the variable a value on another line. If you wanted to write the preceding code on separate lines, you would type

```
var score;  
score = 0;
```

Declaring a Data Type

An excellent way to check for errors in your code is to tell Flash what type of data a variable will hold. This is called *declaring a data type*. That way, if you put a number in a variable that is supposed to hold text, Flash will tell you that you made a mistake in your code. This will help a lot when you start writing large blocks of code. To declare a data type for a variable, type a colon after the variable name and then the data type of the variable. The data type for a number is called Number, the data type for text is called String, and the data type for true or false is called Boolean. If you wanted to create a variable called myName with a value of "Todd" (or your own name), and you wanted to tell Flash that variable has a String data type, you would type

```
var myName:String = "Todd";
```

NOTE

The value of a string can be any combination of letters, spaces, special characters, and numbers. When you give the value of a string, you put the value in quotes. Code in quotes has a green color, so string values can be easily recognized among the other elements of your code.

There are many different data types, and I will discuss more of them later on.

EXERCISE 2-1: Creating Variables

Now you will get some practice creating variables.

1. Open Flash and create a new Flash file. If you are working in Flash CS3, make sure the Flash file is an ActionScript 2.0 file.
2. Select the first keyframe of Layer 1 and open the Actions panel by pressing F9 (Windows) or OPTION F9 (Mac).
3. In the Actions panel, make sure Script Assist is turned off and the Toolbox is hidden.
4. In the Scripting pane, create a variable by typing

```
var
```



Notice the word `var` is blue. That's because `var` is a keyword in Flash. *Keywords* are reserved ActionScript words that have special meaning. When you create a variable, avoid choosing names that turn blue after you type them. This could create errors that keep your code from running properly.

5. Type a space, and then name the variable `myName`. Your code should look like this:

```
var myName
```

6. Give the variable a data type of `String`. Notice that the word `String` turns blue, because it is a keyword in Flash. Your code should look like this:

```
var myName:String
```

7. Set the `myName` variable's value to your name. Remember that the value of a string should be in quotes. Notice that the quotes and the value within the quotes turn green. Green is the default color for string values in Flash. Your code should look like the code next, with your name instead of `Todd`.

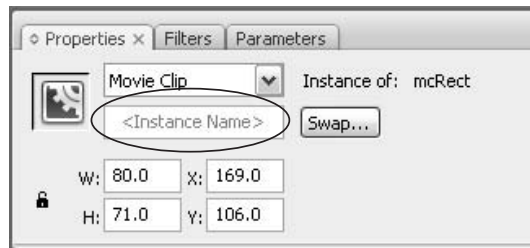
```
var myName:String = "Todd";
```

8. Close the file. You do not need to save your changes.

UNDERSTANDING INSTANCE NAMES

In order to change values and control movie clips using ActionScript, the movie clip you want to control needs an *instance name*. A movie clip's instance name makes it unique from other copies (or instances) of the same symbol (the item stored within

FIGURE 2-2



The Instance Name field in the Property Inspector

the Library). To give a movie clip instance (or a button instance) an instance name, select the instance on the Stage and type in the Instance Name field in the Property Inspector (shown in Figure 2-2). When naming instances, follow the same rules you follow when creating variables.



If you end movie clip instance names with `_mc`, Flash will give you access to code hinting, which will make typing your code much easier.

WORKING WITH PROPERTIES

Properties are variables that are attached to objects. You are already familiar with properties. When you create a movie clip and change its X or Y position, you are modifying its properties. You can also modify an object's properties using ActionScript. To modify an object's properties with ActionScript, you use something called *dot syntax*. In ActionScript, dot syntax is used to communicate to something within something else. For example, the property that controls the X position of a movie clip is called `_x`. If you wanted to modify the X position of a movie clip with an instance name of `my_mc` and set it equal to 10, you would type

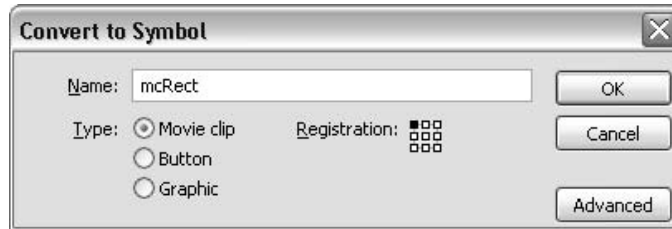
```
my_mc._x = 10;
```

Here, the dot in the code (after `my_mc`) tells Flash to look for something called `_x` inside of `my_mc`. Then, you can give a value to the property using the equal sign just as when you give a value to a variable.

EXERCISE 2-2: Modifying Movie Clip Properties Using ActionScript

In this exercise, you will control some different properties of a movie clip by using ActionScript.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. Using the Rectangle tool, draw a rectangle on the Stage.
3. With the Selection tool, select the rectangle and convert it to a movie clip by using the keyboard shortcut F8.
4. Name the movie clip **mcRect**, and make its registration point at the top left, as shown in the following illustration. When you are done, click OK.



5. Make sure you are in the Main Timeline by clicking Scene 1 at the bottom of the Timeline.
6. Select the movie clip on the Stage and, in the Instance Name field in the Property Inspector, type **rect_mc** (as shown in Figure 2-3).

NOTE

Always remember to give instance names to any symbols you plan to modify using ActionScript.

7. Create a new layer and name it **actions**. Make sure the actions layer is at the top.

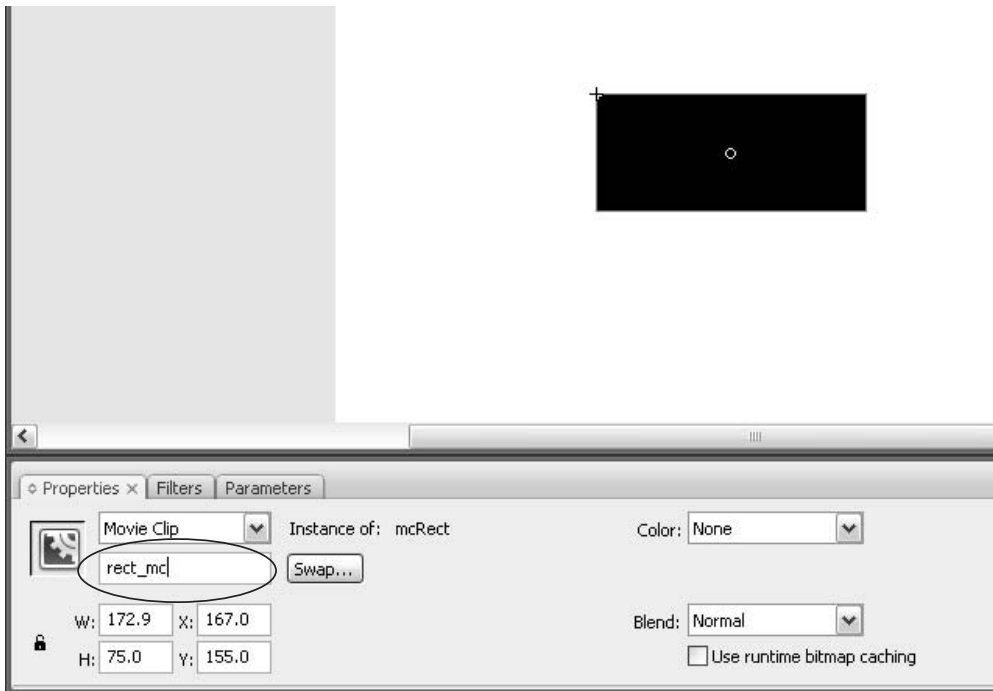
TIP

Having your code in a layer at the top is a best practice and makes your code easy to maintain.

8. Select the first keyframe of the actions layer and open the Actions panel by pressing F9 (Windows) or OPTION F9 (Mac).
9. In the Actions panel, click in the Scripting pane and type

```
rect_mc._x = 10;
```

FIGURE 2-3



The Instance Name field in the Property Inspector

TIP Because this movie clip has an instance name ending with `_mc`, Flash gives you code hinting when you type a dot. Code hinting is Flash's way of helping you write ActionScript code. This shows you properties you can modify on a movie clip. The properties you can modify begin with an underscore (`_`).

- 10.** If you look at the movie clip on the Stage, you will notice its X position does not update. That is because the ActionScript you type does not run until the Flash movie is playing.
- 11.** Press `CTRL-ENTER` (Windows) or `COMMAND-RETURN` (Mac) to test the movie. Notice the `rect_mc` instance moves 10 pixels from the left edge of the Stage. The values of properties you set in code override the values on the Stage. Nice!
- 12.** Close the file. You do not need to save your changes.

WORKING WITH NUMBERS

In ActionScript, performing basic mathematical operations is simple. The following table outlines some of the basic mathematical operators in Flash.

Symbol	Use	Example
+	Addition	$1 + 2 = 3$
-	Subtraction	$2 - 1 = 1$
*	Multiplication	$1 * 2 = 2$
/	Division	$2 / 1 = 2$

There are also a few other ways to work with numbers in ActionScript. The following table shows a few examples of this.

Symbol	Use	Example	Result
+=	Add to a number's current value	$a += 3$	$a + 3$
--	Subtract from a number's current value	$a -= 3$	$a - 3$
*=	Multiply by a number's current value	$a *= 3$	$a * 3$
/=	Divide by a number's current value	$a /= 3$	$a / 3$
++	Increment by 1	$a ++$	$a + 1$
--	Decrement by 1	$a --$	$a - 1$

WORKING WITH STRINGS

When working with strings, plus signs (+) concatenate, or connect, two strings. The following code demonstrates two strings being connected:

```
var firstName:String = "Todd";
var lastName:String = "Perkins";
var fullName:String = firstName + lastName;
// fullName is "ToddPerkins"
```

In order to create spaces when using concatenation, you need to concatenate a space. This is demonstrated in the following code in the var fullName line:

```
var firstName:String = "Todd";
var lastName:String = "Perkins";
var fullName:String = firstName + " " + lastName;
// fullName is "Todd Perkins"
```

You will see more examples of string concatenation as you start creating games.

UNDERSTANDING FUNCTIONS AND METHODS

When you create Wii Flash games, you will write many lines of code. Functions allow you to reuse parts of your code so that you don't have to keep writing the same lines over and over again. Methods are also functions, but they are functions attached to objects. I will talk more about methods in the next section.

Using Simple Functions

If you have experience working with Flash before reading this book, you may already be familiar with functions. Functions are needed to control the playback of a Flash movie. If you wanted a Flash movie to stop playing instead of looping endlessly, you would use the stop function. To run a function, type the function name and opening and closing parentheses. If running the function is the end of your ActionScript statement, type a semicolon after the parentheses. The code to run the stop function and make your Flash movie stop playing is

```
stop();
```

You can place this code at any keyframe in your actions layer and Flash will stop the movie when the Playhead reaches that frame.

EXERCISE 2-3: Using the stop Function

In this exercise, you will learn how to use the stop function to stop the playback of your Flash movies.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. Draw a rectangle on the Stage.
3. Select the rectangle and convert it to a movie clip.

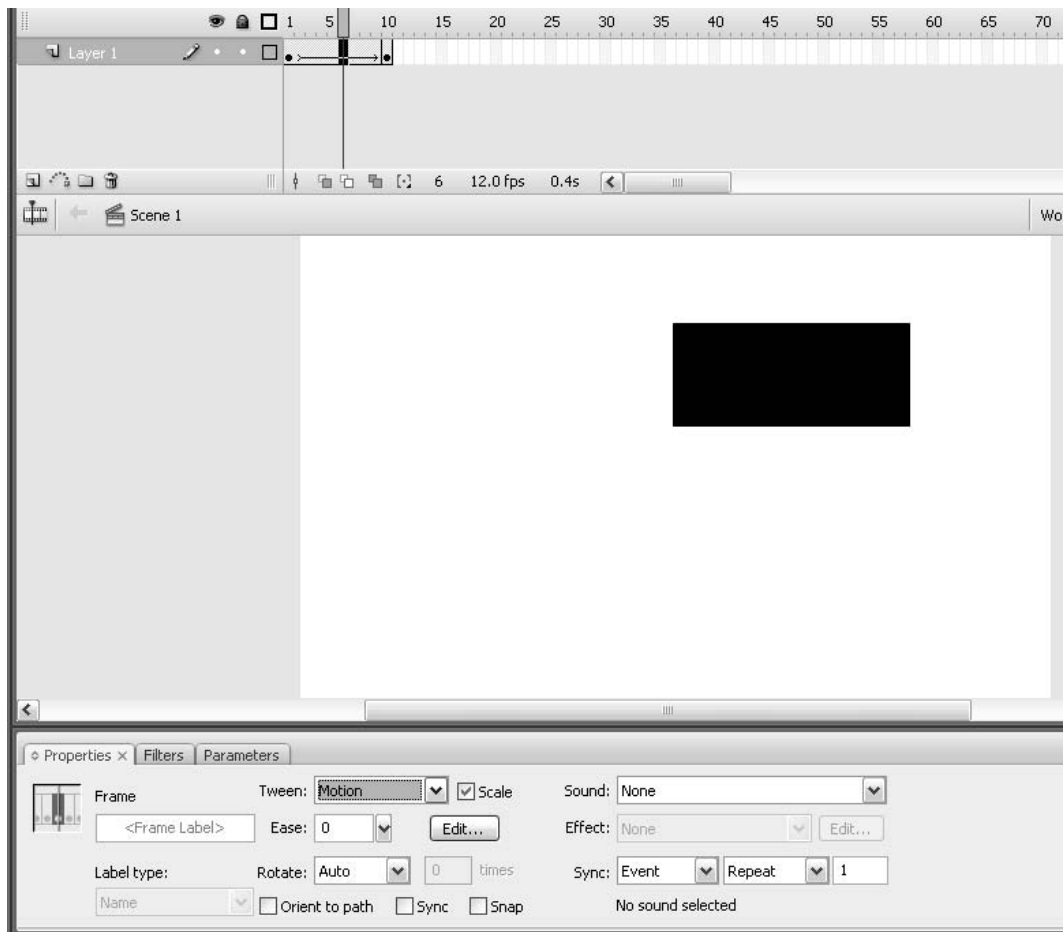
NOTE

Though this exercise does not require the movie clip to have a particular name, you should still avoid giving this movie clip a name with spaces or special characters. If you don't remember what characters are special characters, you can think of special characters as the characters that are used to represent curse words (%&#!, etc.). Like many people, Flash might get upset if you swear at it.

4. In the Main Timeline, select frame 10 in the Timeline and press F6 on your keyboard to create a keyframe.

5. In frame 10, select the movie clip on the Stage and move it to a different area of the Stage. In the next step, you will create a motion tween.
6. In the Timeline, select any frame between frame 1 and frame 10, and in the Property Inspector, set the Tween type to Motion (as shown in Figure 2-4).
7. Test the movie and preview the animation. Notice the animation loops endlessly. In the next steps, you will use ActionScript to stop the movie when it reaches the last frame. Close the preview window when you are done.

FIGURE 2-4



Creating the motion tween



The keyboard shortcut to test the movie is CTRL-ENTER (Windows) or COMMAND-RETURN (Mac).

- 8.** Create a new layer above Layer 1 and name the new layer **actions**.
- 9.** In the actions layer, select frame 10 and press F7 on your keyboard to create a blank keyframe.
- 10.** Select frame 10 of the actions layer and open the Actions panel by pressing F9 (Windows) or OPTION-F9 (Mac).
- 11.** In the Scripting pane of the Actions panel, type

```
stop();
```
- 12.** Test the movie. Watch the movie play and stop on the last frame. Cool!



You do not have to save the file in order to preview the ActionScript you wrote.

- 13.** Close the file. You do not need to save your changes.

Defining and Using Custom Functions

Up to this point, I have only discussed the code to run a function. Defining a function refers to the area of code where you tell Flash a function's name and what it does. Running a function refers to using a function that either you created or is built into Flash. Follow these steps to create, or declare, a custom function:

- 1.** To create a function, you need to use the function keyword. In the same way that the var keyword tells Flash you are going to create a variable, the function keyword tells Flash you are going to create a function.
- 2.** After the function keyword, you need to type a space and give your function a name. Function names have the same restrictions as variable and instance names.
- 3.** After the function name, type opening and closing parentheses. These correspond to the opening and closing parentheses used when you are running the function.
- 4.** After the parentheses, type a colon and the type of data the function will return. I will talk more about functions returning values later in this section. If your function does not return a value (and many of the functions you create will not), the return data type is Void. This step is optional, but it is a best practice to include it.

5. The last step in creating a function is to tell Flash what the function does. After all of the code in the previous steps, place the code that runs when you run the function in opening and closing curly braces ({}). These can be on the same line or on separate lines, although it may be easier for you to read your code if the brackets are on their own lines.

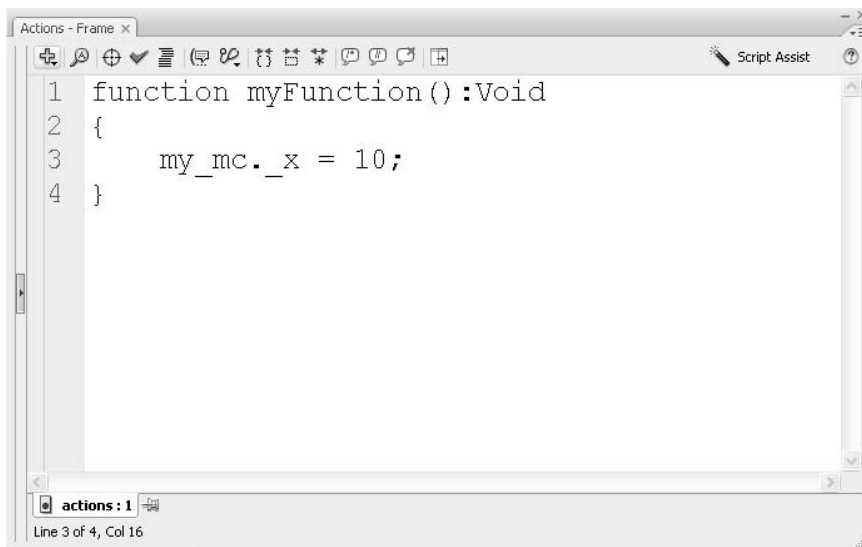
A function called `myFunction` that sets the X position of a movie clip called `my_mc` to 10 looks like Figure 2-5.

To run a custom function that you created, type the function name and opening and closing parentheses. Figure 2-6 shows Flash running a function.

NOTE It doesn't matter where you write the code to run the function as long as it is defined in the same frame as when you run it. When you run a function, Flash will search for the function definition in the same frame where you run the function. If there is a definition for that function, Flash will run the function.

You can create variables inside the curly braces of a function, but the variable is known and can only be used inside of that function. This concept will make more sense once you start creating games, and I will discuss it in greater detail then.

FIGURE 2-5

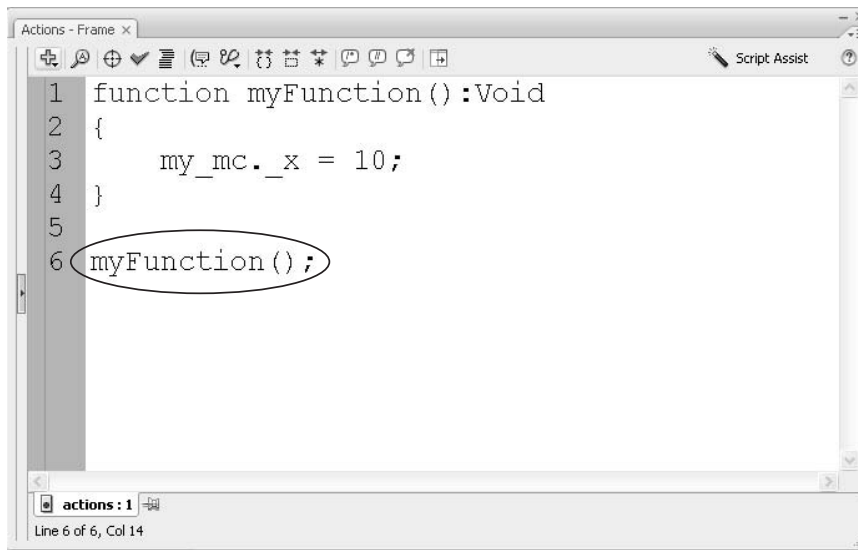


```
1 function myFunction():Void
2 {
3     my_mc._x = 10;
4 }
```

The screenshot shows the 'Script Assist' window in the Flash IDE. The window title is 'Actions - Frame x'. The code editor contains the following code:
1 function myFunction():Void
2 {
3 my_mc._x = 10;
4 }
The status bar at the bottom indicates 'actions : 1' and 'Line 3 of 4, Col 16'.

A simple function called `myFunction`

FIGURE 2-6



Running a function called myFunction

EXERCISE 2-4: Creating and Using a Custom Function

In this exercise, you will create and use a simple custom function.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. On the Stage, draw a rectangle with a black fill and convert it to a movie clip named mcRect.
3. Give the rectangle an instance name of **rect_mc**.
4. Create a new layer named **actions**. Make sure it is at the top.
5. Select the first keyframe of the actions layer and open the Actions panel.
6. In the Actions panel, click in the Scripting pane and type

```
function setValues():Void
{
}
}
```

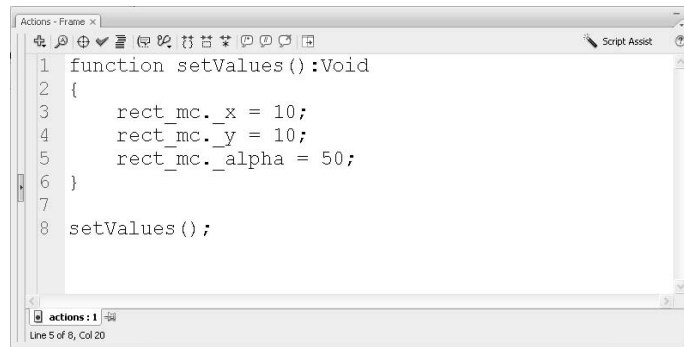
7. Inside the curly braces of the `setValues` function, type

```
rect_mc._x = 10;  
rect_mc._y = 10;  
rect_mc._alpha = 50;
```

NOTE The `_y` property controls a movie clip's Y position, and the `_alpha` property controls the opacity of a movie clip from 0 to 100 (0 is invisible, 100 is fully opaque).

8. Below the closing curly brace of the `setValues` function, run the `setValues` function by typing

```
setValues();
```



```
1 function setValues():Void  
2 {  
3     rect_mc._x = 10;  
4     rect_mc._y = 10;  
5     rect_mc._alpha = 50;  
6 }  
7  
8 setValues();
```

actions: 1
Line 5 of 8, Col 20

9. Test the movie. Notice in the preview window, shown in the following illustration, that the rectangle is 10 pixels from the left edge of the Stage, 10 pixels from the top edge of the Stage, and semitransparent. Nice!



10. Save the file, and keep it open for the next exercise.

Understanding Parameters

At the beginning of this section I mentioned that functions are reusable blocks of code. One way to make a function reusable is by using something called *parameters*. Parameters are values that can change each time you use a function.

To understand parameters, let's take a look at some of Flash's built-in functions that use parameters. If you want your Flash movie to navigate to a certain frame when you click a button, you would use the `gotoAndPlay` or `gotoAndStop` function to control the navigation of your movie. If you wanted your movie to play from the first frame, you would type

```
gotoAndPlay(1);
```

In this example, the number 1 is a parameter passed in to the `gotoAndPlay` function. Each time you use the `gotoAndPlay` function, you can pass in a different parameter, depending on the frame you want to play. You will use the `gotoAndPlay` function often when you start creating games.

NOTE

The `gotoAndStop` function works the same as the `gotoAndPlay` function, but when the movie gets to the frame you pass in, the movie stops instead of playing.

Another function that uses a parameter is called `trace`. The `trace` function (also known as `trace` statement) is used when writing code to test if certain parts of the code are working. When you use the `trace` function, the value you pass in shows up in the Output window when you test the movie. The Output window, shown in Figure 2-7, is used to show you `trace` statement messages and errors when testing your Flash movies. For example, if you wanted to write a note to yourself to appear in the Output window by using the `trace` function, you could type

```
trace("note to self: Hi self!!!");
```

When you `trace` a string, the message shows up in the Output window exactly as you typed it. You can also use a `trace` statement to check the value of a variable. If you created a variable called `myName` and gave it a string value of "Todd", you could

FIGURE 2-7



Trace statement showing up in the Output window

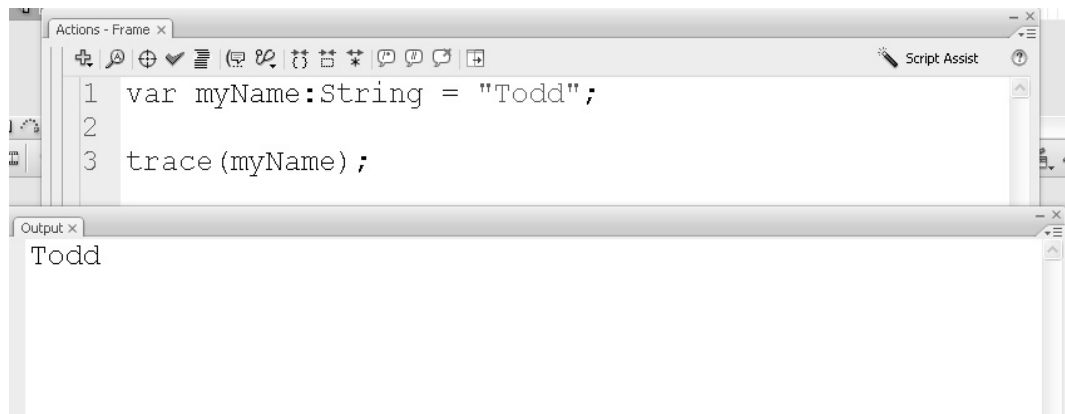
`trace myName` and the value of the variable would show up in the Output window (see Figure 2-8).

Some functions have multiple parameters. If a custom or prebuilt function has multiple parameters, the values are all passed in the parentheses when you run the function, and they are separated by commas.

Creating Functions with Parameters

Making functions that accept parameters is only a little more complex than making functions that do not accept parameters. To make a function accept parameters, you need to do two things: create the parameters when you define the function, and pass in the parameters when you run the function.

FIGURE 2-8



Tracing the value of a variable

Creating parameters inside a function is similar to declaring variables without giving them values. The only difference is that you declare the variables inside of the parentheses of the function declaration, separated by commas instead of semicolons. If you had a function called `myFunction`, and you wanted the function to accept a parameter called `myParam`, that had a data type of `String`, you would type

```
function myFunction(myParam:String):Void
{
}
}
```

Once you have defined a function that accepts parameters, you need to pass in parameters when you run the function. The parameters you pass in must have the same data type you specified when you defined the function. For example, if you wanted to run the function defined in the code in the last paragraph, you would type

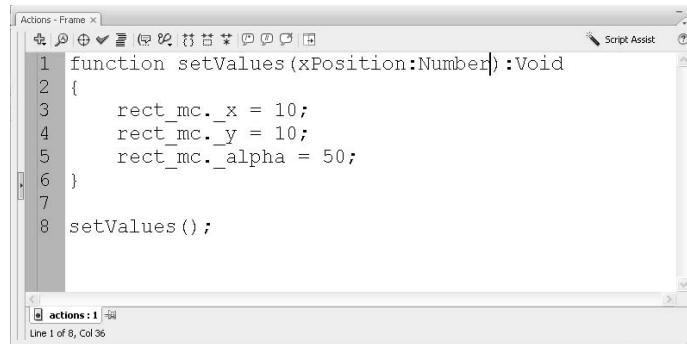
```
myFunction("Any value will work, as long as it is a String!");
```

Notice the value passed in is a string (you can tell because it's in quotes). Why does it have to be a string? It has to be a string because the function is looking for a string to give a value to the `myParam` parameter. Take a minute to look over the code and think about how it works.

EXERCISE 2-5: Writing and Using Functions with Parameters

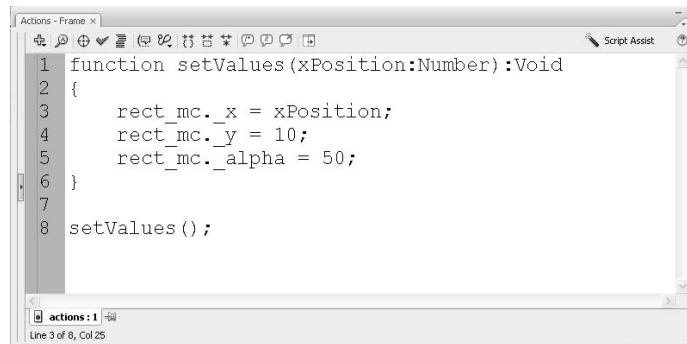
In this exercise, you will get some practice creating functions that accept parameters.

1. Open the file you worked with in the last exercise.
2. Select the first keyframe of the actions layer and open the Actions panel.
3. Inside the parentheses of the `setValues` function you declared at the top of the code, create a parameter called `xPosition` with a `Number` data type.



```
1 function setValues(xPosition:Number):Void
2 {
3     rect_mc._x = 10;
4     rect_mc._y = 10;
5     rect_mc._alpha = 50;
6 }
7
8 setValues();
```

4. Inside the curly braces of the `setValues` function, find the line where you set the `_x` property of `rect_mc` to 10 (shown in the following illustration), and replace the 10 with `xPosition`.

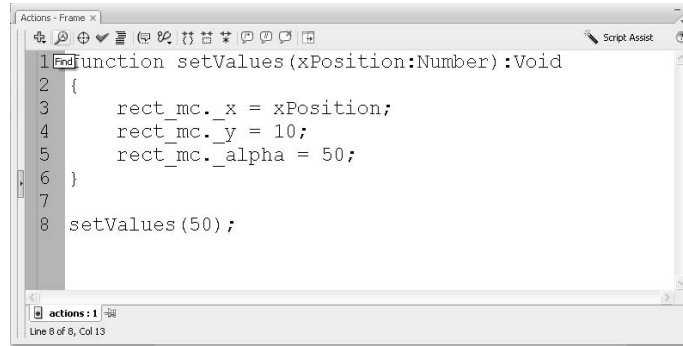


```
1 function setValues(xPosition:Number):Void
2 {
3     rect_mc._x = xPosition;
4     rect_mc._y = 10;
5     rect_mc._alpha = 50;
6 }
7
8 setValues();
```

NOTE

At this point, the `xPosition` parameter does not have a value. The value of the parameter will be whatever value you pass in when you run the `setValues` function.

5. At the bottom of the code, find the line where you run the `setValues` function. Inside the parentheses, type `50`, as shown next. The number that you type here will be the X position of the `rect_mc` movie clip when you preview the movie.



```
1 function setValues(xPosition:Number):Void
2 {
3     rect_mc._x = xPosition;
4     rect_mc._y = 10;
5     rect_mc._alpha = 50;
6 }
7
8 setValues(50);
```

The screenshot shows a window titled "Actions - Frame x" with a "Script Assist" toolbar. The code editor contains the following code: `1 function setValues(xPosition:Number):Void`, `2 {`, `3 rect_mc._x = xPosition;`, `4 rect_mc._y = 10;`, `5 rect_mc._alpha = 50;`, `6 }`, `7`, and `8 setValues(50);`. The status bar at the bottom indicates "actions : 1" and "Line 8 of 8, Col 13".

6. Test the movie. Notice the `rect_mc` movie clip is 50 pixels from the left edge of the Stage, just as you passed in when you ran the `setValues` function. Nice!
7. Close the preview window, and change the value passed into the `setValues` function. Take a minute to review how the code is working, and test the movie again.
8. When you are finished, close the file. You do not need to save your changes.

Understanding Return Values

Some functions give values back after they run. This is called *returning a value*. The `Math` class, or `Math` object, has many properties and methods that perform various mathematical operations. Many of the `Math` methods are functions that return values. One example of a function that returns a value is the `Math.random` method. This function is attached to the `Math` object, so it is called a method instead of a function. The `Math.random` method gives you a random number between 0 and 1, but not including 1. You would see a random decimal number in the Output window if you typed this code and tested the movie:

```
trace(Math.random());
```

NOTE

Notice this line of code ends with two closing parentheses. The first one is for the `Math.random` method, and the second is to close the `trace` statement.

When the `Math.random` method runs, Flash does some work behind the scenes and returns a number. Once you start creating games, you will use and create many different functions that return values.

Creating Functions That Return Values

Creating functions that return values takes two steps. First, you tell Flash what type of data a function will return. Then, inside the curly braces where you define the function, you use the `return` keyword to return the data.

Setting a Return Data Type

The return data type is set after the colon in the first line of the definition of a function. Up to this point, you have not created functions that return values, so you have used `Void` as the return data type. You can set the return data type by replacing `Void` with a type of data (i.e., `String`, `Number`, `Boolean`, etc.). In the code that follows, the function `myFunction` has a return data type of `String`.

```
function myFunction():String
{
}
}
```

Returning a Value

To make a function return a value, you must use the `return` keyword inside the function definition. In the example that follows, the function `myFunction` returns a string value of "Hello!".

```
function myFunction():String
{
    return "Hello!";
}
}
```

It's important to note that when a function has a return data type, you must also return the data inside the function using the `return` keyword. The type of data returned must match the return data type.

EXERCISE 2-6: Creating a Function with Return Data

In this exercise, you will get some practice creating functions that return data, and see how that data can be used.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. Change the name of Layer 1 to **actions**.
3. Select the first keyframe of the actions layer and open the Actions panel.
4. In the Scripting pane, create a function called `sayHello` that returns a string by typing

```
function sayHello():String
{

}
```

NOTE

In the preceding code, `String` is capitalized because it's referring to a data type. This is also the case with all other data types.

5. In order for this function to return a string, you must use the `return` keyword (all lowercase) inside the function. The `return` keyword should turn blue if it is typed correctly. Make this function return "Hello!" The code should look like this:

```
function sayHello():String
{
    return "Hello!";
}
```

6. Run this function inside a `trace` statement to make sure it is working. Make sure the `trace` statement is below the closing curly brace of the function definition. The code should look like this:

```
function sayHello():String
{
    return "Hello!";
}
trace(sayHello());
```

7. Test the movie, and notice the message that appears in the Output window. When you are finished, close the preview window.
8. Now, you will make this function reusable by having it accept a parameter. Create a parameter called `person` with a data type of `String`. The code should look like this:

```
function sayHello(person:String):String
{
    return "Hello!";
}
```

```
}  
trace(sayHello());
```

- 9.** In the trace statement, pass in a person's name as a string to the sayHello function. The code should look like the code that follows, but the name does not have to be Todd.

```
function sayHello(person:String):String  
{  
    return "Hello!";  
}  
trace(sayHello("Todd"));
```

- 10.** Inside the sayHello function, place the person parameter in the return line using concatenation. Remember to put a space after "Hello", to tell Flash there should be a space between "Hello" and the person's name. The code should look like this:

```
function sayHello(person:String):String  
{  
    return "Hello " + person + "!";  
}  
trace(sayHello("Todd"));
```

- 11.** Before you test the movie, take a minute to look at how the code is working. Ask yourself the following questions: How is the name in the trace statement being sent to the sayHello function? What is the return line of code doing with the person parameter? What will happen when I test the movie?
- 12.** Test the movie and watch what displays in the Output window.
- 13.** When you are finished, close the file. You do not need to save your changes.

Working with Methods

As I mentioned earlier, *methods* are functions that are attached to objects. To use an object's methods, type the object name, a dot, and the name of the method, with the appropriate parameters passed in parentheses. One method available to use on movie clips is the stop method. If you wanted to stop the playback of a movie clip instance called my_mc, you would type the following code:

```
my_mc.stop();
```

You will work more with methods and learn about different methods later on, when you start creating games.

WORKING WITH EVENTS AND EVENT HANDLERS

We are going to work with events and event handlers often when creating Wii Flash games. *Events* are things that happen, such as when a button gets clicked. *Event handlers* are special functions that run when an event takes place, instead of running when you specify in the code. You do not need to write code for events to take place, but you need to create event handlers if you want something to happen when an event occurs.

Writing Event Handlers

Because event handlers are functions, much of the syntax for creating them is identical to that of creating functions. Most of the event handlers you will be working with do not have parameters or return values. When creating an event handler, you need to tell Flash the object that received the event, state the name of the event, and set that equal to a function. Setting something equal to a function is something you haven't done at this point. The syntax for setting something equal to a function is a little bit different from using the function keyword first. What follows is an example of setting something equal to a function.

Instead of writing

```
function myFunction(parameter:String):Void
{
}

```

you would write

```
myFunction = function(parameter:String):Void
{
}

```

This way of writing functions is acceptable for creating non–event handler functions, but I prefer using this method only when creating event handler functions. That way, I can quickly differentiate between the two.

Here is an example of how an event handler looks in code, with the object being `objectName` and the event being `eventName`:

```
objectName.eventName = function():Void
{
}

```

NOTE The dot after `objectName` tells Flash that `eventName` is connected to `objectName`. Just as objects have properties, as discussed earlier in this chapter, many objects also have events associated with them. Writing an event handler allows you to run code when an event happens. Without an event handler, the event still occurs, but no code will run.

In Flash, almost all events begin with “on.” For example, the event name for clicking a movie clip (or a button) is `onRelease`. If you wanted to run a trace statement when you clicked a movie clip called `my_mc`, you would type

```
my_mc.onRelease = function():Void
{
    trace("you clicked my_mc!");
}
```

There are several events that you will be working with as you create Wii Flash games. Most of the events we will use are connected to movie clips. The following table outlines the movie clip events you will be using most often.

Event Name	Event
<code>onRelease</code>	When the mouse is pressed and released on a movie clip
<code>onPress</code>	When the mouse is pressed down on a movie clip
<code>onRollOver</code>	When the mouse rolls over a movie clip
<code>onRollOut</code>	When the mouse rolls out of a movie clip
<code>onEnterFrame</code>	Runs repeatedly with the frame rate of the Flash movie

EXERCISE 2-7: Writing Event Handlers

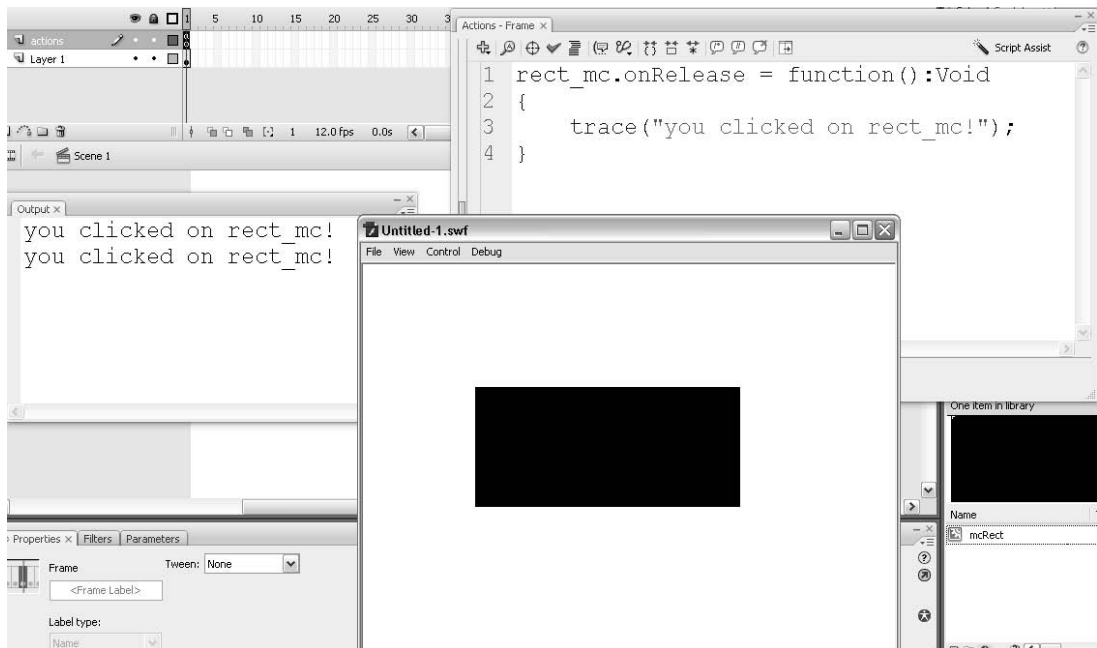
In this exercise, you will write an event handler that reacts to a mouse click.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. On the Stage, draw a rectangle and convert it to a movie clip named **mcRect**.
3. In the Main Timeline, select the movie clip on the Stage, and type **rect_mc** for its instance name in the Property Inspector.
4. Create a new layer above Layer 1 and name the new layer **actions**.
5. Select the first keyframe of the actions layer and open the Actions panel.
6. In the Actions panel, click in the Scripting pane, and type

```
rect_mc.onRelease = function():Void
{
    trace("you clicked on rect_mc!");
}
```

7. Test the movie. Click the rectangle on the Stage. Notice the message that shows up in the Output window each time you click the button (shown in Figure 2-9). Nice!
8. Try using another event from the last table, and see what happens. Note that if you use the `onEnterFrame` event, you may have trouble closing the Output window because messages continue to appear in it. This is because the `onEnterFrame` event happens as long as the Flash movie is playing. To stop the messages from appearing, close the preview window.
9. Close the file. You do not need to save your changes.

FIGURE 2-9



The working event handler

WORKING WITH CONDITIONAL STATEMENTS

A *conditional statement* is an if/then statement. In other words, you have power to run a block of code only if a certain condition is true. You also have power to run a block of code only if a condition is not true. You can even set up multiple conditions that get checked in a particular order, and have a block of code run for the first condition that is true.

Creating a Conditional Statement

Conditional statements, or if statements, have three parts: the conditional keyword, the condition, and what to do if the condition is true. The condition is placed in parentheses and evaluated as true or false (a Boolean value), and the code that executes if the condition is true is placed in curly braces. The code that follows demonstrates a basic conditional statement:

```
if(condition)
{
    // Run this code if the condition is true
}
```

Using Boolean Variables in a Conditional Statement

When working with Boolean variables in a conditional statement, using the variable name represents a value of true. The code that follows demonstrates a variable called administrator inside a conditional statement:

```
var administrator:Boolean = true;
if(administrator)
{
    //This code will run because administrator is true
}
```

Using Conditional Operators

Sometimes you will want to use a conditional statement to compare two values. You may want to know if a number variable is greater than or less than another number variable. Sometimes you will want to test whether a number is equal to another number. To do this, you need to use conditional operators. Here is a list of the most common conditional operators:

Operator	Meaning	Example
>	<i>Is greater than</i>	<i>if(a > b)</i>
<	<i>Is less than</i>	<i>if(a < b)</i>
==	<i>Is equal to (different from one equal sign used when setting values)</i>	<i>if(a == b)</i>
>=	<i>Is greater than or equal to</i>	<i>if(a >= b)</i>
<=	<i>Is less than or equal to</i>	<i>if(a <= b)</i>
!=	<i>NOT equal to</i>	<i>if(a != b)</i>

The following code example demonstrates a conditional statement using conditional operators:

```
if(3 <= 4)
{
    // This code will run because 3 is less than or equal to 4
}
```

Writing an else Statement

An else statement must follow an if statement, and the code inside the curly braces of an else statement will run only if its accompanying if statement's condition is evaluated as false. Because else statements run when an if statement's condition is false, they do not require a condition. The following code demonstrates an else statement:

```
if(3 > 4)
{
    // This code will not run because the condition is false
}
else
{
    This code will run because the if statement's condition is false
}
```

Writing an else if Statement

If you want to create a hierarchy of conditions, you can use else if statements. In other words, you can have Flash check if a condition is true, and if it is not true check another condition, and if that condition is not true, Flash can check yet another condition. You get the point. To create an else if statement, you must first have an if statement. The following code demonstrates an example of an else if statement:

```
if(condition 1)
{
```

```
// run if condition 1 is true
}
else if(condition 2)
{
// run if condition 1 is false and condition 2 is true
}
else if(condition 3)
{
// run if condition 1 and 2 are false and condition 3 is true
}
else
{
// run if conditions 1, 2, and 3 are false
}
```

Writing Compound Conditional Statements

Conditions in a conditional statement can be broken up into expressions. You can check if multiple expressions are true within one condition. A conditional statement with more than one expression is also called a *compound* conditional statement. In a compound conditional statement, you can check if one expression and another expression are both true, or check if one expression or another expression is true. The conditional operator for “and” is two ampersands (&&). The conditional operator for “or” is two vertical pipes (||).

NOTE

Vertical pipes are created by holding SHIFT and pressing backslash (\) on the keyboard.

The following code demonstrates compound conditional statements.

```
if(a > 0 && a < 5)
{
// This will run if a is greater than zero AND less than five
}

if(a == 0 || a == 5)
{
// This code will run if a is equal to zero OR if a is equal to five
}
```

EXERCISE 2-8: Writing Conditional Statements

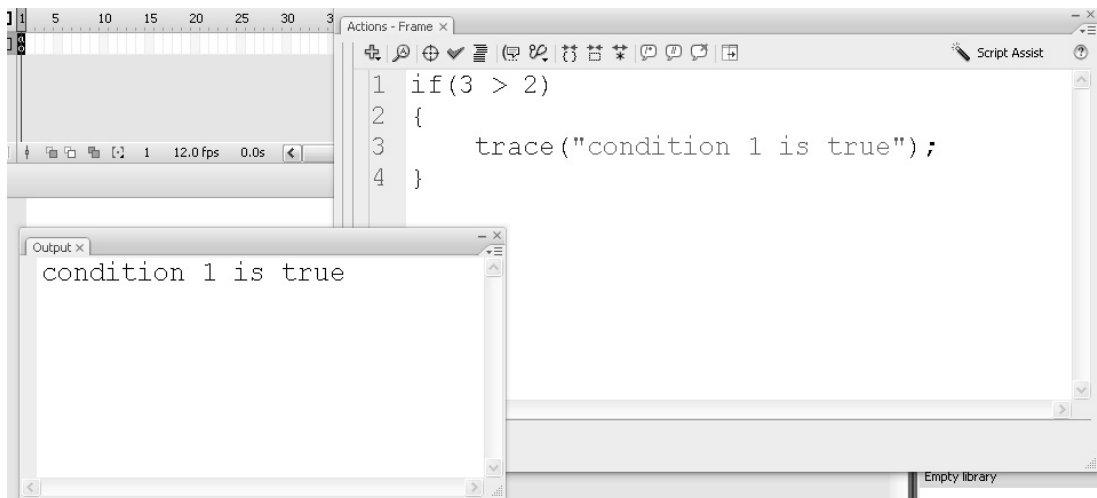
In this exercise, you'll get some practice writing conditional statements yourself.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. Change the name of Layer 1 to **actions**.
3. Select the first keyframe of the actions layer and open the Actions panel.
4. In the Actions panel, click in the Scripting pane and create a conditional statement that evaluates whether 3 is greater than 2 by typing the following code:

```
if( 3 > 2)
{
    trace("condition 1 is true");
}
```

5. Test the movie. Notice the message, shown in Figure 2-10, that comes up in the Output window.

FIGURE 2-10



The trace statement appearing in the Output window

6. Change the code in the if statement to read if 3 is less than 2, so the condition will be false. Your code should match this snippet:

```
if( 3 < 2)
{
    trace("condition 1 is true");
}
```

7. Below the if statement, write an else statement. Inside the else statement, trace “all conditions are false”. Your code should look like this:

```
if( 3 < 2)
{
    trace("condition 1 is true");
}
else
{
    trace("all conditions are false");
}
```

8. Test the movie. Notice the message in the Output window indicating all conditions are false. Close the preview window when you are finished.
9. Between the if statement and the else statement, create a compound else if statement that is true if 3 is less than 2 OR if 3 is greater than 2. Inside the else if statement, trace “condition 2 is true”. Your code should look like this:

```
if( 3 < 2)
{
    trace("condition 1 is true");
}
else if( 3 < 2 || 3 > 2)
{
    trace("condition 2 is true");
}
else
{
    trace("all conditions are false");
}
```

10. Test the movie. Notice the message in the Output window indicating condition 2 is true. Close the preview window and look over the code again to make sure you understand how it is working.
11. Close the file. You do not need to save your changes.

UNDERSTANDING LOOPS

A *loop* is a block of code that runs a certain number of times. Loops help you to perform tasks where you repeat yourself. Using loops, it is just as easy to run a block of code one time as it is to run a block of code a thousand times. You will use loops often when you create games later on.

Writing a For Loop

There are a few different types of loops, but the one you will be working with most is called a *for* loop. The following code shows a basic for loop that will run 20 times.

```
for(var i:Number = 0; i < 20; i ++)  
{  
    // this is the code that is looped  
}
```

A *for* loop begins with the *for* keyword. Then, in parentheses there are three elements, separated by semicolons.

The first element is the loop variable. In this case, there is a variable called *i* that is a number data type with a value of zero. This number is known as the loop index. The loop index can be used inside the loop. I will talk about using the loop index later.

The second element states how long the loop will run. The code in the preceding sample will run as long as the *i* variable is less than 20. Because the *i* variable has an initial value of zero, the loop will run 20 times. If you tested the movie with this code in your actions layer, you would see the numbers 0 through 19 in your Output window. This is because the variable *i* has an initial value of 0, and the loop stops running once *i* is equal to 20. Each time a loop runs is called an iteration.

The third element tells Flash what to do if the condition in the middle is not true after the block of code in the curly braces runs. In this case, the *i* variable will increment by 1.

It is important to note that a loop runs a certain number of times, but it is instantaneous in actual time. In other words, if you used a loop that ran 20 times, and added 1 to the X position of a movie clip each time the loop ran, you would not see the movie clip animate. Instead, you would see the movie clip 20 pixels from its starting point.

EXERCISE 2-9: Creating a For Loop

In this exercise, you will create a simple for loop to see a loop in action.

1. Open Flash and create a new Flash file (ActionScript 2.0).

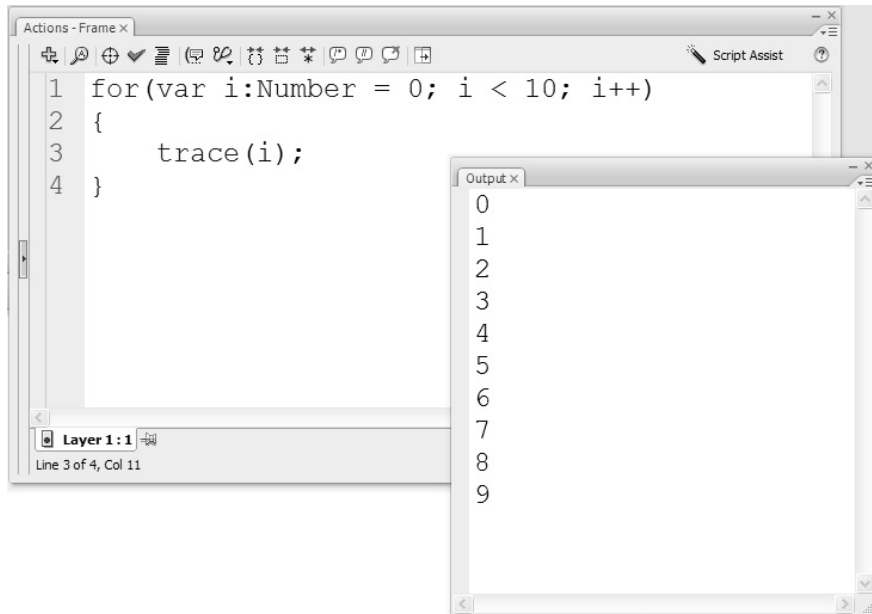
2. Change the name of Layer 1 to **actions**.
3. Select the first keyframe of the actions layer and open the Actions panel.
4. In the Scripting pane, create a loop that runs 10 times by typing this code:

```
for(var i:Number = 0; i < 10; i++)
{
}
```

5. Inside the curly braces of the loop, trace the value of *i*. Your code should match what is shown here:

```
for(var i:Number = 0; i < 10; i++)
{
    trace(i);
}
```

6. Test the movie. Notice the values that appear in the Output window, shown in the illustration, are 0 through 9.



7. Close the file. You do not need to save your changes.

In the next section, I will discuss arrays, and how to use a loop to capture data from an array.

UNDERSTANDING ARRAYS

Arrays are a little different than anything you've worked with up to this point, and you'll use them often when creating games. Earlier in this chapter, I discussed how variables are containers that hold data. *Arrays* are containers that can hold *multiple* data values.

Creating an Array

The Array data type is also called a class. A *class* refers to the behavior of an object. You already have experience working with classes. Classes are similar to symbols. When you create an instance of a movie clip, you are actually creating an instance of the MovieClip class. To create an instance of a class that is not visible, like the Array class, you type the keyword “new,” a space, and the name of the class followed by parentheses. Once you create an instance of the Array class, you can access its properties and methods in the same way you access the properties and methods of a movie clip. The following code shows an example of creating a new instance of the Array class called `my_ary`.

```
var my_ary:Array = new Array();
```

As discussed earlier, arrays hold multiple data values. The values of an array are stored in numerical order. Items in an array can be referenced by their numerical position, which in an array is called an index number. The first index of an array is always zero, and array indices are always referenced using square brackets (`[]`).

Adding to an Array

There are a few different ways to add data to an array. One way is to set the value of a particular index of an array. The following code demonstrates the data “Some text” being added to an array called `my_ary` at index 0, and the data “Some more text” added to the array at index 1.

```
var my_ary:Array = new Array();  
my_ary[0] = "Some text";  
my_ary[1] = "Some more text";
```

Data inside an array is not limited to strings. In fact, arrays can hold any type of data, and even different types of data.

You can also create an array and add data to it using one line of code. To do this, instead of using the `new` keyword to create an array, you type square brackets and the values of the array, separated by commas. The following code demonstrates some

numeric values being added to an array called `my_ary` on the same line the array is created.

```
var my_ary:Array = [7,3,15,6,12];
```

Another way to add data to an array is to use the `push` method of the `Array` class. The `push` method adds an item to the last index of an array. If the array has no indices, the data is placed at index zero. The following code shows the `push` method adding the data “Pushed text” to an array instance called `my_ary` at index zero.

```
var my_ary:Array = new Array();  
my_ary.push("Pushed text");
```

Capturing Data from an Array

There are several different ways to capture data from an array. One way is to reference the data by its index in the array. The code that follows demonstrates tracing data at index one of an array called `my_ary`.

```
var my_ary:Array = [7,3,15,6,12];  
trace(my_ary[1]);  
// Traced value is 3
```

In this code, the value in the Output window would show 3, because the first value in the array, which is 7 in this case, is at index zero.

Another helpful tool for working with arrays is the `length` property. The `length` property of an array gives you the number of items in the array (not the last index number). You can then use this information in a code loop (or a `for` loop) to use the data in each index of an array, which you will do often when you create games. The following code shows a loop that runs for each index of the array `my_ary` and traces the value of each index.

```
var my_ary:Array = [7,3,15,6,12];  
for(var i:Number = 0; i < my_ary.length; i++)  
{  
    trace(my_ary[i]);  
}  
// The Output window would then show each of the values in the array
```

EXERCISE 2-10: Working with Arrays

In this exercise, you will get some practice working with arrays by creating a simple array and tracing different values of data in that array.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. Change the name of Layer 1 to **actions**, select the first keyframe of the actions layer, and open the Actions panel.
3. In the Scripting pane, create an array called `my_ary` and give the array some text values by typing the following code:

```
var my_ary:Array = ["text1", "text2", "text3", "text4"];
```

4. Add another value to the array using the push method. Your code should look similar to this:

```
var my_ary:Array = ["text1", "text2", "text3", "text4"];  
my_ary.push("text5");
```

5. Below all the code you have written, trace the value of index 4 of the array by typing the name of the array and placing 4 inside square brackets. Your code should look similar to this:

```
var my_ary:Array = ["text1", "text2", "text3", "text4"];  
my_ary.push("text5");  
trace(my_ary[4]);
```

6. Test the movie. Notice the value that appears in the Output window matches the fifth item in the array, which is index 4. With the preceding code, the Output window would show “text5”.
7. Close the window and the Flash file. You do not need to save your changes.

UNDERSTANDING DOT SYNTAX AND COMMUNICATION

Dot syntax is Flash’s method of communication. You have actually been using dot syntax throughout this chapter to set property values and use methods. To reference a property or a method attached to an object, you typed the object name and a dot (.) and then the property or method name. You can also use dot syntax to communicate between different objects, like movie clips.

Understanding Communication Between Objects

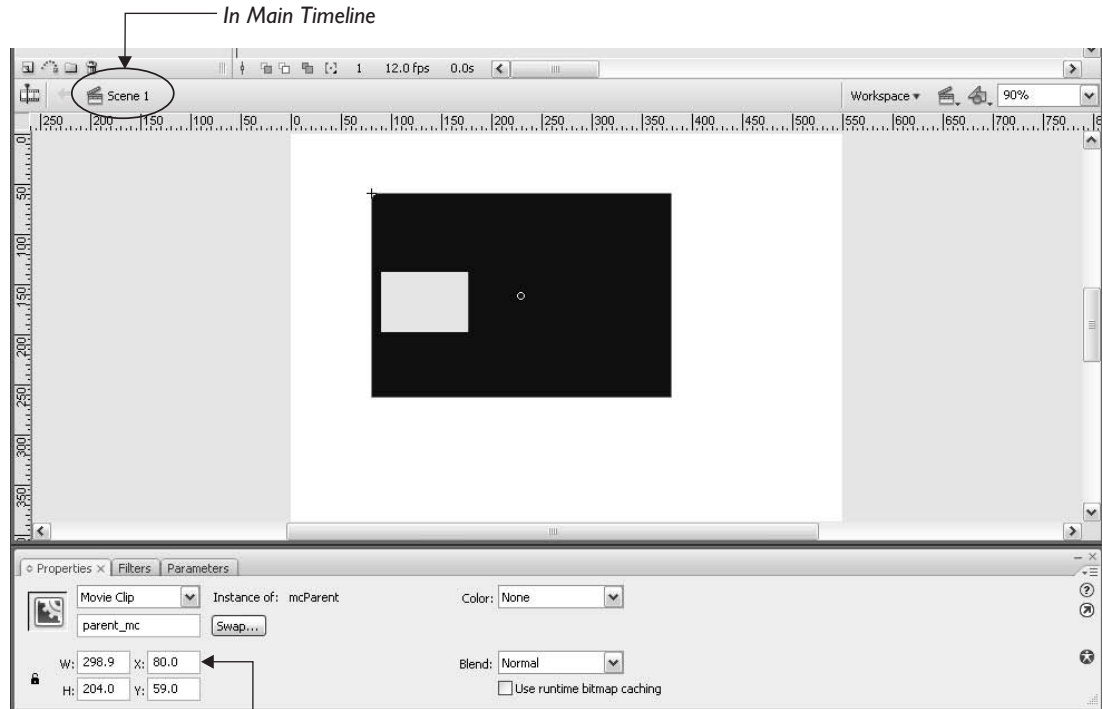
In the last chapter, I discussed how movie clips can be placed in other movie clips. Using dot syntax, it is possible to modify properties or use methods of any movie clip with an instance name, even if that movie clip is nested within another movie clip. When a movie clip is nested within another movie clip, the child movie clip is treated

as a property of the parent movie clip. The following code demonstrates an example of modifying the X position of the movie clip `child_mc` that is within the movie clip `parent_mc`.

```
parent_mc.child_mc._x = 10;
```

Here, the X position of the `child_mc` movie clip is set to 10. It's important to note that positioning of children inside a parent movie clip is relative to the parent movie clip, as shown in Figures 2-11 and 2-12.

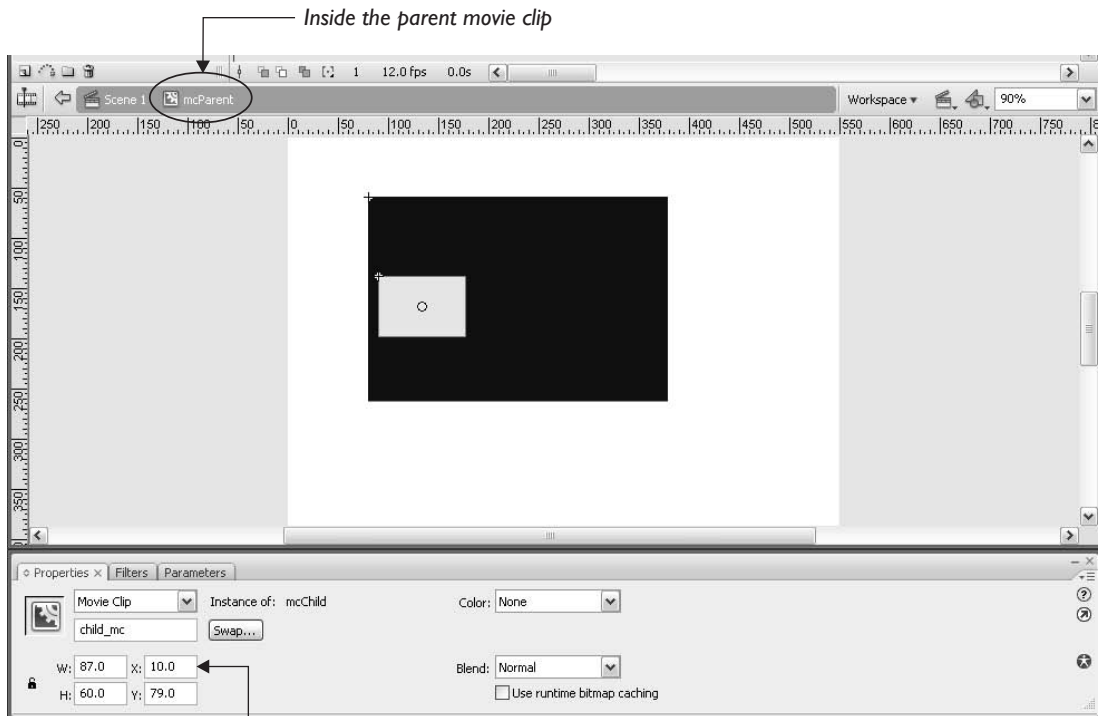
FIGURE 2-11



X position of `parent_mc` is 80.

Positioning of a parent movie clip

FIGURE 2-12



X position of child movie clip is 10, even though it is further to the right than the parent Movie Clip.

Positioning of a child movie clip

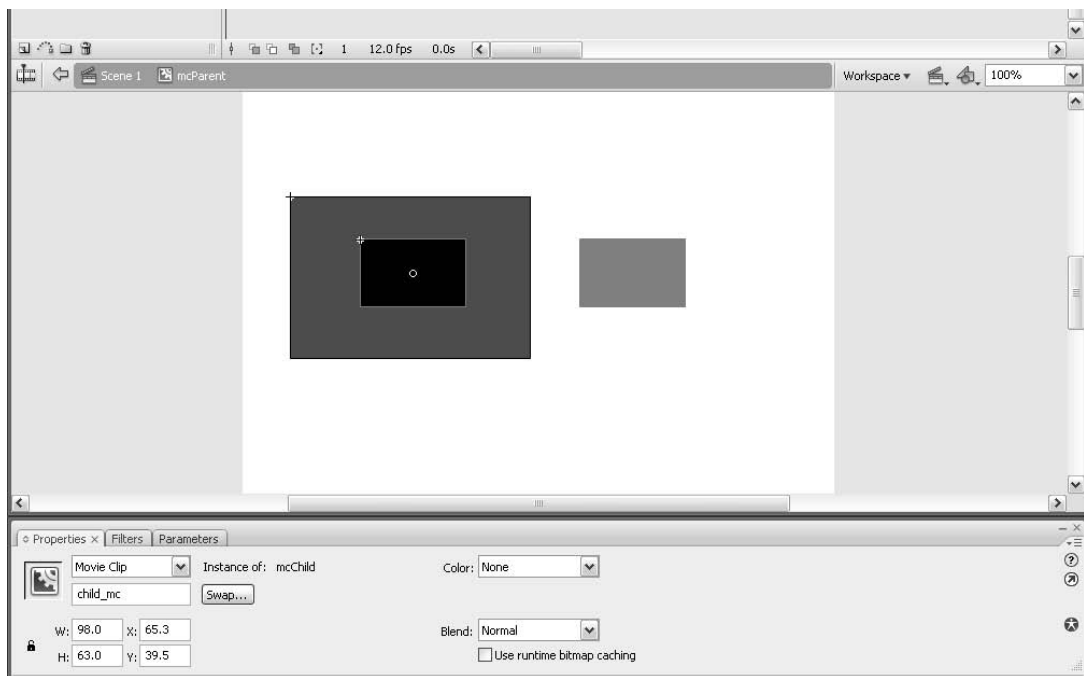
EXERCISE 2-11: Communicating to a Child Movie Clip

In this exercise, you will get practice communicating between movie clips. Understanding this concept is key to using ActionScript 2.0.

1. Open Flash and create a new Flash file (ActionScript 2.0).
2. On the Stage, draw two rectangles with different fills, one larger than the other, and turn them into movie clips. Name the larger one **mcParent** and name the smaller one **mcChild**. Make sure to give both symbols top-left registration.

3. Enter the Timeline of mcParent by double-clicking the parent movie clip on the Stage. Make sure that you see mcParent in the Timeline, indicating you are editing the mcParent movie clip.
4. Inside the Timeline of the mcParent movie clip, drag an instance of mcChild onto the Stage, placing it in the center of the larger rectangle. Give the mcChild movie clip an instance name of `child_mc`. Note the X position of `child_mc`, shown in Figure 2-13. You will change this using ActionScript in a few steps.
5. Return to the Main Timeline by clicking Scene 1 in the Timeline.
6. On the Stage, select the mcParent movie clip and give it an instance name of `parent_mc`.
7. In the Main Timeline, create a new layer called `actions`. Lock the actions layer, so you don't accidentally place art on that layer. Select the first keyframe of the actions layer and open the Actions panel.

FIGURE 2-13

The `child_mc` instance inside the mcParent movie clip

8. In the Actions panel, type the following code:

```
parent_mc.child_mc._x = 0;
```

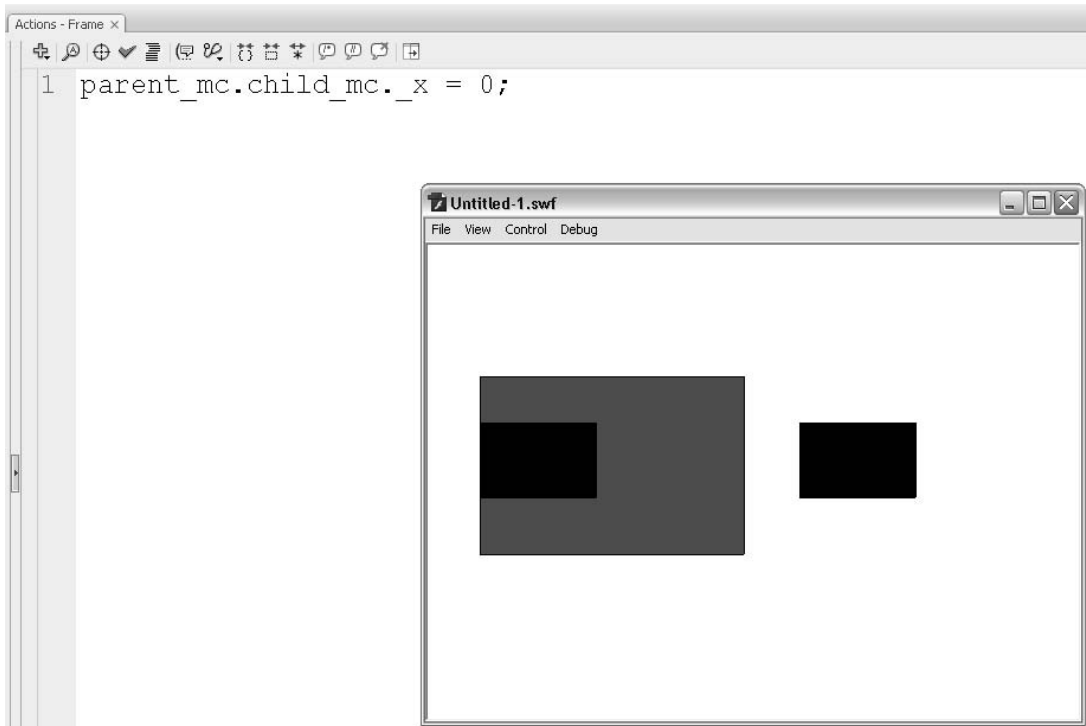
9. Test the Movie. Notice the child_mc movie clip is on the left edge of its parent movie clip, parent_mc, as shown in Figure 2-14. This is because the X position of a child is relative to its parent.
10. Save the file, or keep it open for the next exercise.

Understanding Communication from Children to Parents

Using dot syntax, you can also communicate from a child to a parent. Each movie clip has a property that holds its parent movie clip, called `_parent`. If a movie clip with an instance name of `child_mc` were nested within another movie clip, you could change the X position of the parent movie clip using the following code:

```
child_mc._parent._x = 10;
```

FIGURE 2-14



The `child_mc` movie clip's X position is zero, relative to its parent movie clip, `parent_mc`.

NOTE All children inside a parent movie clip will move along with the parent when the parent movie clip is moved. This is because the parent movie clip acts as a container, and when you move the container, everything within the container moves with it.

One advantage to using the `_parent` property is that you do not have to know the instance name of the parent movie clip. Another advantage that you will see throughout the rest of this book is using `_parent` will make the code you write more reusable.

TIP You can use `_parent` repeatedly to refer to grandparents and great-grandparents of a movie clip and so on. If there were a movie clip named `child_mc`, which was nested within a movie clip called `parent_mc`, which was nested within a movie clip called `grandParent_mc`, which was nested within a movie clip called `greatGrandParent_mc`, you could reference the great-grandparent movie clip from within `parent_mc` by typing `child_mc._parent._parent._parent`. Here, the first `_parent` refers to `parent_mc`, the second to `grandParent_mc`, and the third to `greatGrandParent_mc`.

EXERCISE 2-12: Communicating Between Children and Parents

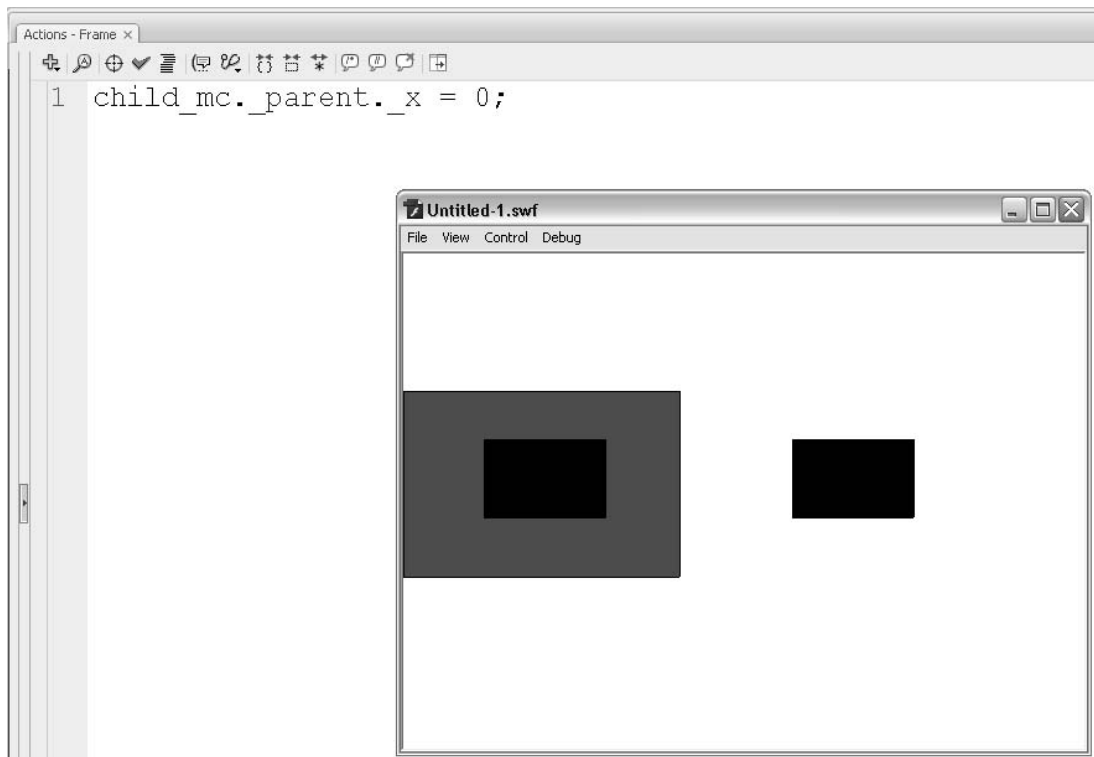
In this exercise, you will get practice communicating from a child movie clip to its parent movie clip using the `_parent` property. You will also look at placing ActionScript inside of a movie clip's timeline.

1. Open the file you created in the last exercise.
2. On the Main Timeline, select the first keyframe of the actions layer, open the Actions panel, and delete the code you wrote.
3. Double-click the `parent_mc` movie clip on the Stage to enter its Timeline.
4. Inside the `mcParent` movie clip, create a new layer called **actions**. Select the first keyframe of the actions layer and open the Actions panel.
5. In the Actions panel, type the following code:

```
child_mc._parent._x = 0;
```

6. Test the movie. Notice the parent movie clip is moved to the left of the Stage, and the `child_mc` movie clip is still in its same location relative to the parent movie clip, as shown in Figure 2-15.
7. Take a minute to look over your code and make sure you understand how the code is working. Note that using the name `_parent` is a property of `child_mc`, and is different than the instance name of the parent movie clip

FIGURE 2-15



Positioning of the parent_mc movie clip after testing the movie

on the Main Timeline, which is parent_mc. Using the `_parent` property does not require you to know the name of the parent movie clip.

- 8.** Save the file, or keep it open for the next exercise.

Understanding `_root`

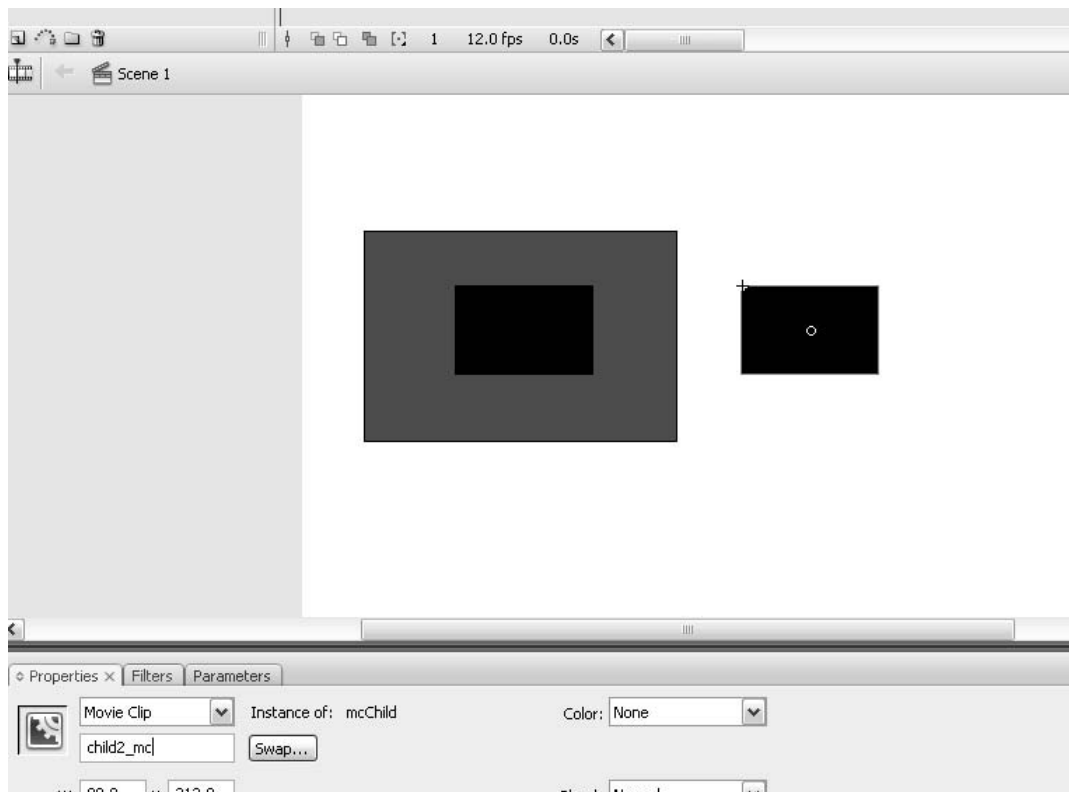
The name `_root` refers to a very useful tool in ActionScript, because it represents the Main Timeline. In fact, `_root`, or the Main Timeline, is actually a movie clip that acts as a container for your entire Flash movie. Understanding that the Main Timeline is a movie clip will be of more use later on. If you place code inside of a movie clip, no matter how many other movie clips that movie clip is nested within, you can use `_root` to reference the Main Timeline. This is a very effective way to communicate from one movie clip to another.

EXERCISE 2-13: Communicating Using `_root`

In this exercise, you will get practice communicating between movie clips using `_root`.

1. Open the file you worked with in the last exercise.
2. In the Main Timeline (or Scene 1), select the unnamed instance of `mcChild` on the Stage and give it an instance name of `child2_mc` (as shown in Figure 2-16).
3. Double-click the movie clip parent `_mc` on the Stage to enter its Timeline.
4. Inside the `mcParent` movie clip, select the first keyframe of the actions layer and open the Actions panel.

FIGURE 2-16

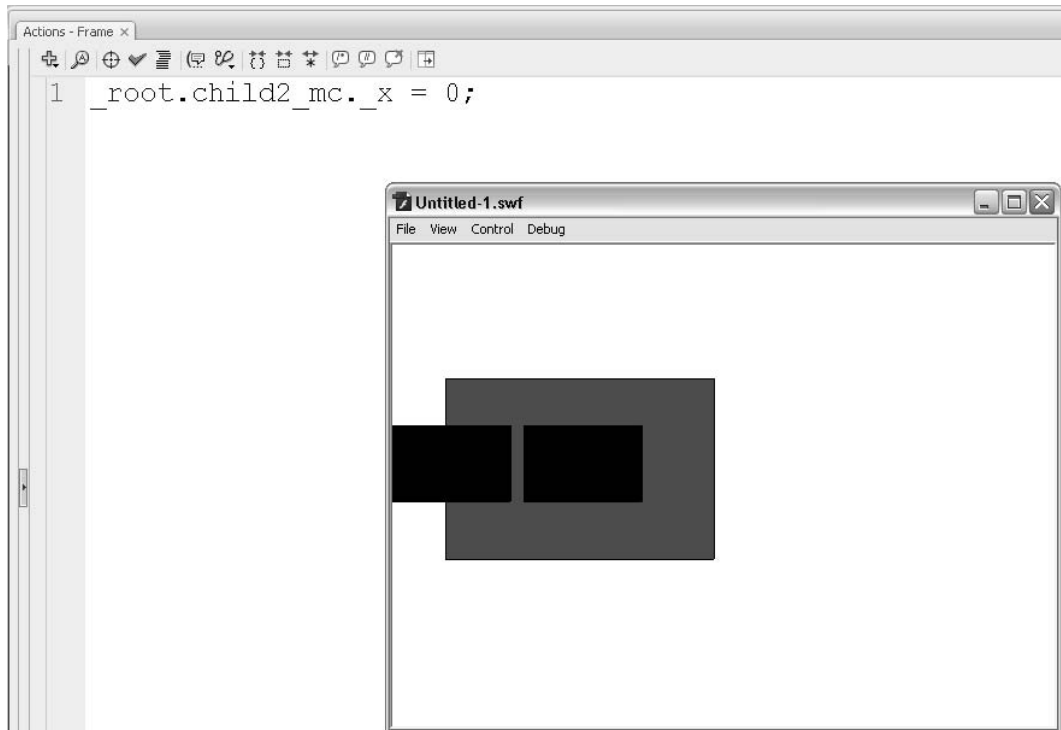


Naming the instance of `mcChild` on the Main Timeline

5. In the Actions panel, delete all the code and replace it with the following code:

```
_root.child2_mc._x = 0;
```
6. Test the movie. Notice the position of the movie clip `child2_mc` is at the left edge of the Stage, as shown in Figure 2-17. Because `_root` references the Main Timeline, you can use this method to communicate to an object on the Main Timeline. You can even use this technique along with the other communication techniques you have learned up to this point to communicate to any object from any object, as long as the objects you are communicating through have instance names.
7. Save the file, or keep it open for the next exercise.

FIGURE 2-17



The movie clip `child2_mc` is at the left edge of the Stage.

Understanding the this Keyword

One of the most challenging concepts to grasp when using ActionScript is the meaning of the “this” keyword. The “this” keyword is relative; its meaning changes, depending on where it is in the code. It refers to the object where the this keyword resides. For example, on the Main Timeline, the this keyword represents the Main Timeline. Inside a movie clip, the this keyword represents that movie clip. If the this keyword is inside an event handler attached to a movie clip, the this keyword represents the movie clip to which the event handler is attached.

Using the this Keyword on the Main Timeline

If you use the this keyword on the Main Timeline, outside an event handler, “this” refers to the Main Timeline. For example, the following code, when the movie plays, would give you a value of `_level0`, which is the same as `_root`.

```
trace(this);
```

Using the this Keyword Inside a Movie Clip

If you use the this keyword inside the Timeline of a movie clip, this refers to each instance of that movie clip. For example, if you placed the following code inside a frame in a movie clip’s Timeline, you would see `_level0`, a dot, and then the instance name of the movie clip.

```
trace(this);
```

Using the this Keyword in an Event Handler

There is one massive trap with understanding the this keyword. When you write an event handler, or any type of function attached to an object, the this keyword refers to the object on which the function is being created. For example, the code that follows shows an `onRelease` event handler for a movie clip called `my_mc` that traces the value of this.

```
my_mc.onRelease = function():Void
{
    trace(this);
}
```

Here, the value in the Output window will show the movie clip `my_mc`. This is because the code inside the curly braces is connected to `my_mc`. Remembering this will help you greatly in working with the this keyword.

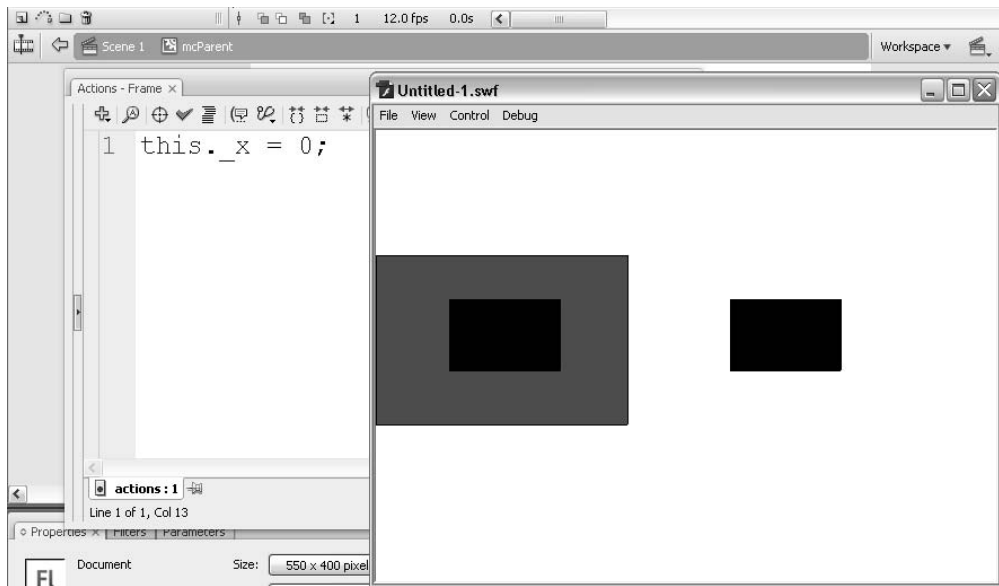
EXERCISE 2-14: Using the this Keyword

In this exercise, you will get practice using the `this` keyword to communicate to different movie clips. You will also get practice using the `this` keyword in different contexts.

1. Open the Flash file you used in the last exercise.
2. On the Main Timeline, double-click `parent_mc` to enter its Timeline.
3. Select the first keyframe of the actions layer, open the Actions panel, and delete the code that you have previously written.
4. Type the following code in the Actions panel:

```
this._x = 0;
```

5. Test the movie. Notice the `parent_mc` movie clip's X position is at the left edge of the Stage, as shown in the following illustration. In this case, the `this` keyword represents `parent_mc`.

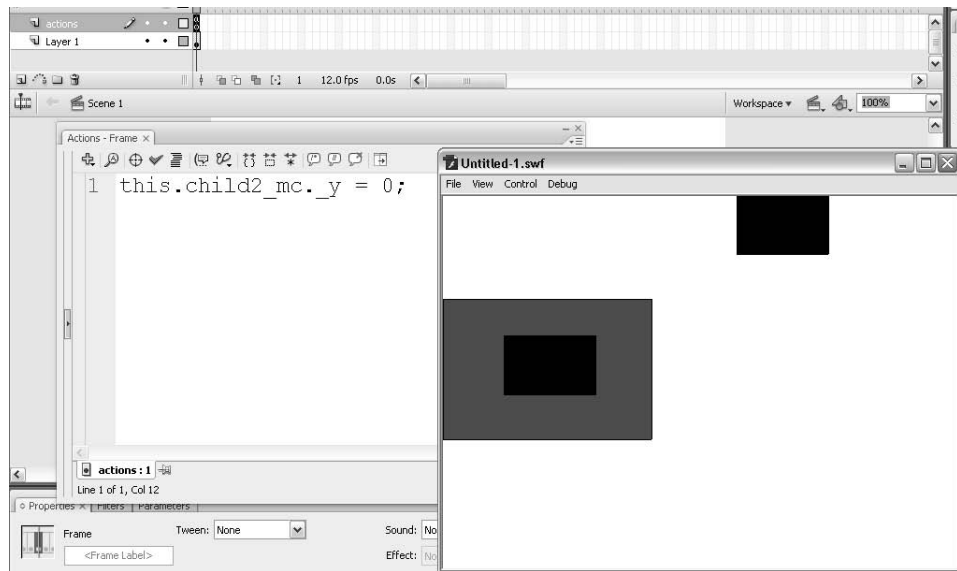


NOTE

The `this` keyword, when placed within a movie clip, represents each instance of that movie clip. If you were to place many instances of `mcParent` on the Main Timeline, each instance would have an X position of zero when the movie is played. Code within a movie clip affects all instances of that movie clip because it is a change made directly on a symbol via Symbol Editing mode, rather than to one particular instance.

6. Return to the Main Timeline by clicking Scene 1 in the Timeline.
7. On the Main Timeline, select the first keyframe of the actions layer and open the Actions panel.
8. In the Actions panel, type the following code:

```
this.child2_mc._y = 0;
```
9. Test the movie. You should then see the child2_mc movie clip at a Y position of zero, as shown next. This is because the this keyword, in this instance, refers to the Main Timeline, and the child2_mc movie clip is on the Main Timeline. This code would look the same if the this keyword were replaced by _root, because here, both represent the Main Timeline.



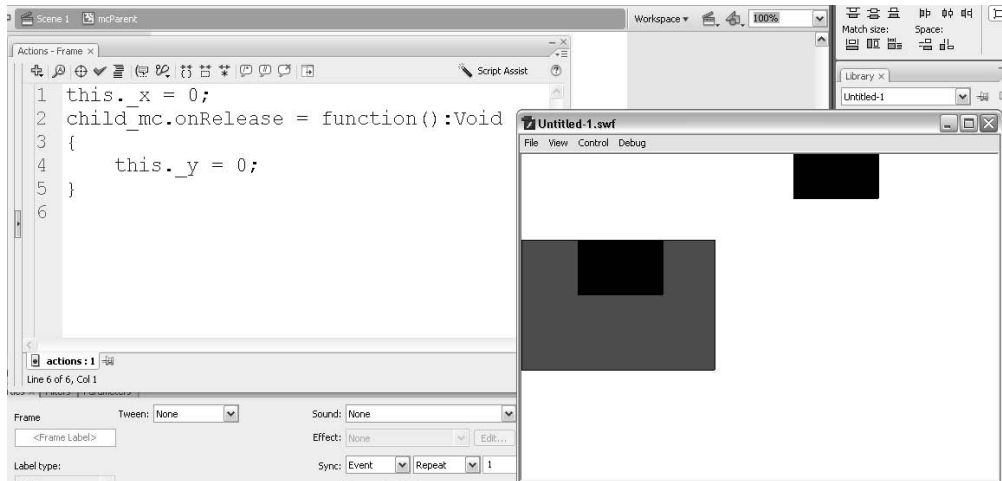
10. On the Stage of the Main Timeline, double-click the parent_mc movie clip to enter its Timeline.
11. Inside the Timeline of mcParent, select the first keyframe of the actions layer and open the Actions panel.
12. In the Actions panel, beneath the code you already wrote, create an onRelease event handler for the child_mc movie clip by typing the following code:

```
child_mc.onRelease = function():Void
{
}
}
```

- 13.** Inside the curly braces of the event handler, type `this._y = 0;` . Your code should match the code shown:

```
child_mc.onRelease = function():Void
{
    this._y = 0;
}
```

- 14.** Before you test the movie, predict what will happen. What does the `this` keyword mean here?
- 15.** Test the movie. Click the `child_mc` movie clip inside the `parent_mc` movie clip. Notice that the child movie clip moves to a Y position of zero, as in the following illustration, showing that in the code you just wrote, the `this` keyword represents `child_mc`.



- 16.** Examine all the code you wrote in this exercise. Notice that the `this` keyword represents three different objects. Make sure you understand which object the `this` keyword represents in each context and why.
- 17.** Save the file, or keep it open for the next exercise.

Using the Insert Target Path Button

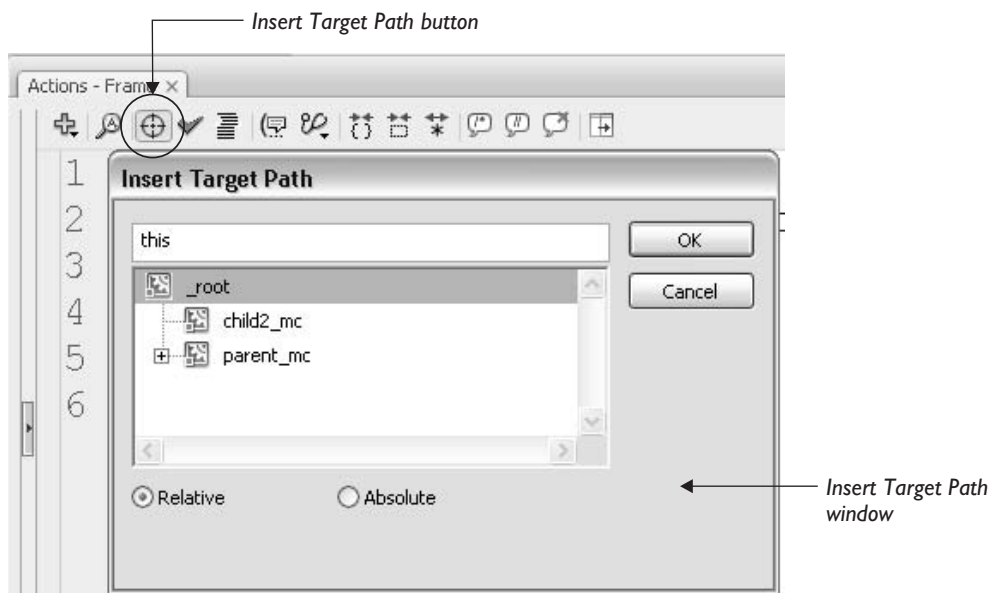
If the syntax to communicate to a particular movie clip ever seems confusing, you can use the Insert Target Path button in the Actions panel to have Flash write the path for you. The Insert Target Path button will give you the path to any object with an instance name in your Flash file. This is a very quick and effective way to communicate from one object to another, without requiring you to remember each object's instance name.

When you click the Insert Target Path button, you will see the Insert Target Path window (shown in Figure 2-18). In the Insert Target Path window, you will see all the objects in your Flash movie that you can communicate with via ActionScript. You can click the buttons to the left of instances that have nested movie clips to expand or collapse the view inside that movie clip. You can then navigate to the instance you wish to communicate with and click OK. Flash will then give you the path to the instance you want to communicate with.

If you use the Insert Target Path button and see objects in your Flash movie that are in greater than and less than signs, Flash is telling you those objects do not have instance names. If you try to create a path to any of those objects. Flash will prompt you to give instance names to them.

You can choose either the relative or absolute path to a movie clip by clicking the radio buttons at the bottom of the window. However, I recommend never using the relative button under any circumstances because Flash will not use the this keyword properly if you attempt to place a target path inside an event handler. For that reason I recommend always using absolute paths (`_root`) when working with this feature.

FIGURE 2-18

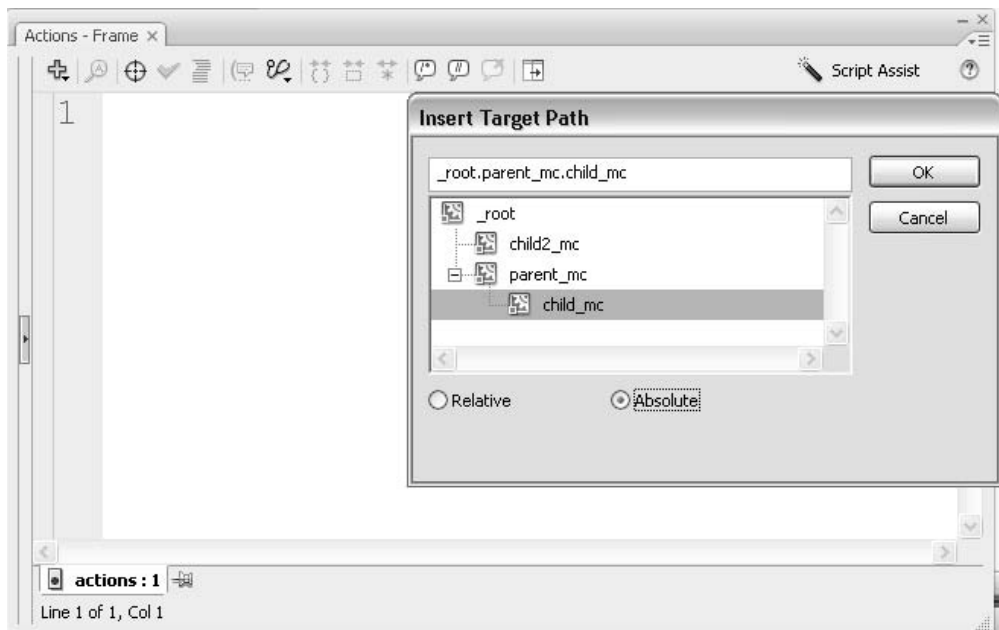


The Insert Target Path button and window

EXERCISE 2-15: Using the Insert Target Path Path Button

In this exercise, you will get practice communicating between movie clips using the Insert Target Path button.

1. Open the file you worked with in the last exercise.
2. On the Main Timeline, select the first keyframe of the actions layer and open the Actions panel.
3. In the Actions panel, delete the code that is already there.
4. Click the Insert Target Path button to open the Insert Target Path window.
5. In the Insert Target Path window, expand the `parent_mc` movie clip by clicking the plus (+) button or the delta button to the left of it. Then, select `child_mc`. Choose the Absolute radio button, as shown here, and click OK.



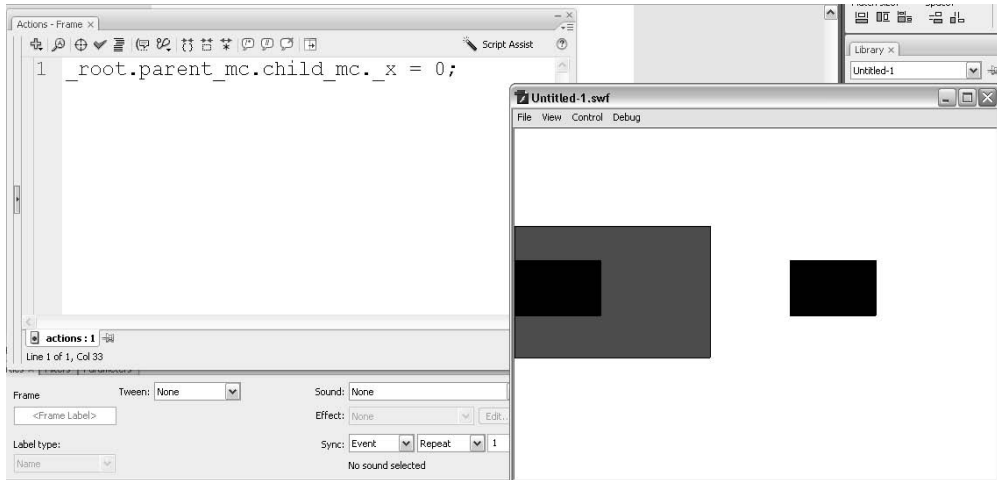
6. Notice that Flash writes out the entire path to `child_mc` for you, including `_root`.

NOTE Here, you do not need `_root` because this code is already on the Main Timeline. Leaving it in the code is completely optional in this case, but if the code were inside a movie clip's Timeline, leaving `_root` would be necessary.

7. After the path Flash wrote for you, set the X position to 0. Your code should match the code shown:

```
_root.parent_mc.child_mc._x = 0;
```

8. Test the movie. Notice the X position of the child_mc movie clip is zero. Nice!



9. Close the file. You do not need to save your changes.

OTHER ACTIONSCRIPT 2.0 RESOURCES

Right now, you know enough ActionScript 2.0 to create simple Wii games. If you want to learn ActionScript 2.0 in greater detail, check out the following resources:

- › **Kirupa.com** This site has loads of free Flash tutorials from basic design to advanced ActionScript. Here, you will also find tutorials in other software and programming languages.
- › **Lynda.com** There are many hours of top-notch ActionScript training here. I highly recommend this site for learning ActionScript, and all kinds of other technologies. You can even watch a lot of content for free!
- › **Actionscript.org** This site is an amazing resource for Flash developers. It's a massive forum where you can find the answer to just about any ActionScript question imaginable.
- › **ChadandToddCast.com** This is my podcast, and here you can find loads of free ActionScript training. You can even send in your questions to be answered in video podcast form.