


introducing ActionScript

LESSON 1

Introductions form the start of any great relationship. Get ready, then, to be introduced to your new best friend: ActionScript! We believe you'll find ActionScript a satisfying companion—especially as you delve deeper into the relationship. Although you may not necessarily think of scripting as a creative endeavor, a working knowledge of ActionScript can spark all kinds of ideas—ones that will enable you to create dynamic content that can interact with your users on myriad levels. Best of all, you'll get the satisfaction of watching your ideas grow into working models that fulfill your projects' objectives.

AS Energy Inc.		Dustin Crail 1234 Main Street Anywhere, AM 12345
This is your current bill. Pay it NOW, or else! Have a nice day!		
Details		Status
Amount you Owe		Message
You currently owe: \$ 60		You have paid your bill in full.
Amount you would like to pay		
Enter the amount you would like to pay: \$	<input type="text" value="60"/>	
Payment		
Make us wealthier by pressing the Pay Now! button		
<input type="button" value="Pay Now!"/>		

We'll plan, create, and test a highly interactive electric-bill payment system.

In this lesson, we'll introduce previous ActionScripters to version 2.0 of the language. For new users, we'll show you some compelling reasons for learning ActionScript, as well as what makes it tick. And if you're feeling a little shy, sometimes the best thing to do is jump right in—which is exactly what you'll do here as you create and test a complete interactive project before lesson's end.

WHAT YOU WILL LEARN

In this lesson, you will:

- Learn about ActionScript 2.0 and how it differs from ActionScript 1.0
- Discover the benefits of learning ActionScript
- Learn to navigate and use the ActionScript editor
- Learn about script elements
- Plan a project
- Write your first script
- Test your script
- Debug your script

APPROXIMATE TIME

This lesson takes approximately one and one half hours to complete.

LESSON FILES

Starting File:

Lesson01/Assets/electricbill1.fl

Completed Projects:

electricbill2.fl

electricbill3.fl

ACTIONSCRIPT MATURES TO VERSION 2.0

If you've been involved with Flash for any length of time, you've seen it evolve from a simple multimedia tool used for creating animated Web graphics and interactive buttons into a multimedia powerhouse that can play external MP3 files, load graphics, play video, talk to a database, and more.

The passionate and innovative community of Flash developers worldwide drives this evolution, to a large degree. By constantly pushing Flash development to new heights, they make us aware not only of the possibilities, but also of the limitations of what can be done.

Fortunately for us, with each new version of Flash, Macromedia strives hard to address these limitations, providing developers with the tools that enable us to do more cool stuff, in the easiest, most efficient manner.

Since its introduction in Flash 5, ActionScript has enabled Flash-based content to soar to new heights, yet there have been obstacles and limitations discovered along the way.

The execution of ActionScript in the Flash 5 player tended to be slow. Tasks that required milliseconds in other scripting/programming languages took seconds with ActionScript. Ask game programmers and they'll tell you that the speed of processing seemed like a lifetime, and it really limited the kind of interactivity that could be used.

In addition to slow processing speeds, ActionScript in Flash 5, while powerful, wasn't very flexible. There wasn't an easy way of implementing object-oriented programming techniques, which enable the creation of more complex and manageable Flash applications.

ActionScript in Flash MX, though not officially dubbed anything more than ActionScript 1.0 with some enhancements, probably could have rightfully been called ActionScript 1.5. It addressed a number of the processing speed issues that plagued Flash 5 ActionScript. In addition, the capabilities of ActionScript in Flash MX were enhanced in ways that enabled the implementation of common object-oriented programming techniques, including the creation of custom object classes and inheritance. While this was definitely a huge step in the right direction, there was still room for improvement.

With Flash MX 2004, Macromedia has introduced ActionScript 2.0. With it come some new capabilities and, more importantly, some new syntax and a new way of structuring and working with your code.

As we discuss in the next section, the changes are somewhat subtle, but they move ActionScript into the realm of a professional-grade programming language. And with what people are demanding from their Flash applications these days, this is definitely a move in the right direction.

NOTE *If you're just learning ActionScript, feel free to skip this section (which probably won't make sense) and move on to the next.*

DIFFERENCES BETWEEN ACTIONSCRIPT 1.0 AND 2.0

If you're familiar with ActionScript 1.0, you'll find that ActionScript 2.0 is similar, yet it has several subtle but important differences. In this section, we examine some of them so that you can take your hard-earned knowledge of ActionScript 1 and put it to use with ActionScript 2.0.

Before we look at these differences, let us first tell you up front that if you want to stick with what you know (programming in ActionScript 1.0), then by all means, do so. ActionScript 1.0 syntax still works in Flash MX 2004. But while this may be the case, and you do have a choice, we recommend you take the time to learn ActionScript 2.0, for a number of reasons.

First, there may come a time when a version of Flash is released that no longer supports ActionScript 1.0. Time spent learning version 2.0 now will be time you might have to spend down the road anyway. And while we can't guarantee it, don't expect ActionScript to jump to version 3.0 any time soon (if ever), because version 2.0 is now built around professional programming language concepts that have stood the test of time. As a matter of fact, outside of a few syntactical differences, writing ActionScript 2.0 code is not much different from writing Java code.

That's right; if you take the time to learn ActionScript 2.0, you'll be able to quickly transition your knowledge to the Java universe, where industrial-strength, cross-platform applications are standard. Consider learning ActionScript: it's a two-for-one deal!

Second, by its very nature and requirements, ActionScript 2.0 will force you to become a more efficient, organized, and better coder. As you will soon see, ActionScript 2.0 has some very strict requirements about how things are done. It's a lot less forgiving than ActionScript 1.0. This may seem like a bad thing, but it actually prevents you from being too sloppy with your scripts, which can make finding bugs a pain, and which can make updating a project months later an arduous task.

Finally, if speed is important to you, you'll be happy to know that ActionScript 2.0 has been shown to be three to seven times faster than ActionScript 1.0. This speed increase will promote the development of even more robust Flash applications.

Let's next look at some of the main differences you'll find between ActionScript 1.0 and ActionScript 2.0.

CASE SENSITIVITY

In ActionScript 1.0, these variable names referenced the same variable:

```
myVariable  
MyVariable
```

In ActionScript 2.0, however, they would be considered two separate variables due to the case difference in their spelling; the first variable begins with a lowercase character, while the second an uppercase character. In fact, all of the following are considered different elements in ActionScript 2.0:

```
myName  
MyName  
MYNAME  
myname  
myNAME
```

This case sensitivity rule applies to all elements in ActionScript 2.0, including instance names, keywords, method names, and so on. Thus, while this syntax will stop all sounds from playing:

```
stopAllSounds();
```

this will cause an error:

```
stopallsounds();
```

TIP *One easy way of testing for case errors is to press the Check Syntax button on the Actions panel. Errors resulting from case mismatches will appear in the output window.*

STRICT DATA TYPING

Variables are used to contain data. This data comes in many forms, including strings of text, numbers, true/false values, references to objects such as movie clip instances, and so on. In ActionScript 1.0, when creating a variable, Flash would automatically assign a data type to it. In this example, Flash understood that the value on the right was of the Number data type:

```
myVariable = 36;
```

While Flash will still automatically recognize this variable as holding a Number data type, ActionScript 2.0 introduces what is known as *strict data typing*, which gives you more control over setting a variable's data type. Here's how it works.

When creating a variable with ActionScript 2.0, you use this syntax not only to create the variable, but also to assign its data type at the same time:

```
var myVariable:Number = 36;
```

As you can see, this syntax is not that different from what you're used to using in ActionScript 1.0. The difference is the addition of the `var` keyword, and the explicit declaration that this variable will hold a number, as indicated by the `:Number` syntax.

A variable that will hold a string looks like this:

```
var myOtherVariable:String = "Hello";
```

In addition to assigning data types to regular variables, strict data typing is used in the creation of object instances:

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

and function definitions:

```
function myFunction (name:String, age:Number)
```

NOTE *Both of these types of strict data typing will be explained in greater depth later in the book.*

How can all those extra keystrokes be an enhancement? Well, there are several ways.

If you're familiar with the Actions panel's ability to provide code hints (which we will discuss shortly), you'll be happy to know that strict data typing activates code hinting for named elements. For example, if you create an Array object using the syntax:

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

the Actions panel will recognize from that point forward that any reference to `myArray` is a reference to an Array object, and it will automatically provide Array object code hints whenever you script that object. In some cases, this new ability eliminates the need for a suffix (`_array`, `_xml`, and `_color`, for example), which was one of the ways that Flash MX enabled code hinting for an object. While the new syntax requires a few more keystrokes, you'll save a few because you no longer have to add suffixes to elements' names, so consider it a decent trade-off.

NOTE *You'll notice we said that in some cases the new ability eliminates the need for suffixes. Visual elements (movie clip instances, buttons, text fields, and components, for example) are not created and named in the same manner as data elements. For this reason, suffixes for these elements' names (such as `_mc`, `_btn`, and `_txt`) are still useful. For this book, we will generally use suffixes only for visual elements.*

Code hinting isn't the only benefit to strict data typing. By indicating the type of data a variable will hold, you save the Flash player a whole lot of time trying to figure it out on its own. When the Flash player is running your application, Flash needs to keep track of the type of data each variable contains. If you use strict data typing, you make this process a lot easier and less processor-intensive for the Flash player. As a result, your scripts will execute much faster. If you want to increase the speed of your applications, use strict data typing.

Finally, strict data typing can be a very simple way of helping you uncover bugs in your code. Here's an example:

```
var favoriteBand:String;
```

This line of script creates a variable named `favoriteBand`, which has been strictly typed to hold a `String`.

Later in your script you assign this variable a value or give it a new one using the syntax:

```
favoriteBand = "The Beatles";
```

However, if somewhere in your script you use something like this syntax:

```
favoriteBand = 46;
```

the Output panel will open and display a “Type mismatch” error when you attempt to export (compile) your movie. This is an error indicating that you've attempted to assign an incorrect value type to a variable. In our example, we've attempted to assign a numeric value (Number data type) to a variable that has been set up to hold a string value.

This functionality can help prevent a number of bugs that can result from simple mistakes you might make when assigning values to variables.

CLASS STRUCTURE

Perhaps the biggest change in `ActionScript 2.0` is the way that object-oriented programming is implemented. For example, with `ActionScript 1.0`, a custom class of objects was defined this way:

```
_global.Person = function (name, age){  
    this.name = name;  
    this.age = age;  
}
```

In ActionScript 2.0, the same class is created using this syntax:

```
class Person {
    var name:String;
    var age:Number;
    function Person (name:String, age:Number){
        this.name = name;
        this.age = age;
    }
}
```

Notice the use of the new `class` keyword. This syntax is very similar to the syntax for creating classes of objects in Java. As you learn more about this syntax (which we won't discuss in detail at this point), you'll quickly see its benefits over ActionScript 1.0.

Another subtle difference in creating custom classes of objects between ActionScript 1.0 and 2.0 is that the script that contains the class definition must exist in its own external **.as** file. This means that the script for defining the `Person` class would need to be saved as an external **.as** file (which is nothing more than a text file with a **.as** file extension) named **Person.as**. This is different from ActionScript 1.0, which allows you to place the code for a class definition on Frame 1 of a movie clip's timeline. This is no longer possible when creating content for the Flash 7 player. Class definitions must exist in an external **.as** file. We'll discuss this functionality in Lesson 7, "Creating Custom Classes." For now, be aware of the difference.

NOTE *You can still place scripts on frames, buttons, etc., but class definitions must exist in their own **.as** files.*

To make the creation of **.as** files as easy as possible, Flash MX 2004 now provides a stand-alone ActionScript editor, which is separate from the Actions panel, and which is used for nothing more than the creation and saving of **.as** files. Later in this lesson, we'll look at both the Actions panel and the stand-alone ActionScript editor.

There are other differences between ActionScript 1.0 and ActionScript 2.0, but this brief overview should provide you with enough insight to get started if you've worked with ActionScript 1.0.

SIMILARITIES BETWEEN ACTIONSCRIPT 1.0 AND 2.0

Okay, up to this point we've only talked about everything that's different between the two versions of ActionScript. But what about them is the same? What hard-earned knowledge of ActionScript 1.0 that you currently have is still applicable to ActionScript 2.0? Fortunately, plenty!

You still control movie clips with this syntax:

```
myMovieClip_mc.gotoAndPlay(15);
```

You still create conditional statements the same way:

```
if (myNumber1 + myNumber2 == 20 && enabled != true){  
    //actions;  
}
```

or looping statements this way:

```
for (i = 0; i <= myVariable; ++i){  
    //actions;  
}
```

Expressions are still created the same way:

```
myVariable = (myNumber1 / 2) + (myNumber2 + 15);
```

And scripts can still be assigned to button and movie clip instances using the `on()` and `onClipEvent()` event handlers.

WHY LEARN ACTIONSCRIPT?

Today, if you're a Flash developer, one thing is certain: animation skills, no matter how phenomenal, are no longer enough. A firm grasp of ActionScript is essential because without it, only the most elementary interactivity is possible. By acquiring an in-depth knowledge of ActionScript, you can:

- Provide a personalized user experience
- Achieve greater control over movie clips and their properties
- Animate elements in your movie programmatically—that is, without using the timeline
- Get data in and out of Flash to create forms, chat programs, and more
- Create dynamic projects that respond to the passage of time or the current date
- Dynamically control sound volume and panning
- Do much more

Add to these benefits the fact that viewing and interacting with Flash content can be more than just a Web experience. Flash can create self-running applications or mini-programs that operate independently of the browser—a capability more people are putting to use to create games, learning applications, and more. If you want to do this, too, you need at least an intermediate knowledge of ActionScript.

ACTIONSCRIPT ELEMENTS

ActionScript is a language that bridges the gap between what you understand and what Flash understands. As such, it allows you to provide both action-oriented instructions (do this) and logic-oriented instructions (analyze this before doing that) in your Flash project. Like all languages, ActionScript contains many different elements, such as words, punctuation, and structure—all of which you must employ properly to get your Flash project to behave the way you want it to. If you don't employ ActionScript correctly, you'll find that interactivity either won't occur or won't work the way you intended. Many of these elements, as well as several other elements such as logical statements and expressions, will be covered in more detail throughout the book.

To begin to understand how ActionScript works, look at this sample script, which contains many of the essential elements that make up a typical script. After the script is a discussion of these elements and their role in the script's execution.

We can assume that this script is attached to a button:

```
on (release) {
    //set the cost of the mug
    var mugCost:Number = 5.00;
    //set the local sales tax percentage
    var taxPercent:Number = .06;
    //determine the dollar amount of tax
    var totalTax:Number = mugCost * taxPercent;
    //determine the total amount of the transaction
    var totalCost:Number = mugCost + totalTax;
    //display a custom message
    myTextBox_txt.text = "The total cost of your transaction is " + totalCost;
    //send the cashRegister_mc movie clip instance to frame 50
    cashRegister_mc.gotoAndPlay (50);
}
```

Although at first glance this may look like Latin, once you become acquainted with some of its elements, you'll understand.

NOTE *Other script elements (for example, objects, functions, loops, properties, and methods) are discussed in detail throughout the book.*

EVENTS

Events occur during the playback of a movie and trigger the execution of a particular script. In our sample script, the event that triggers the script is `on (release)`. This event signifies that when the button to which this script is attached is released, the script will execute. Every script is triggered by an event, and your movie can react to numerous events—everything from a button being pressed to text changing in a text field to a sound completing its playback, and more. We will discuss events in depth in Lesson 2, “Using Event Handlers.”

ACTIONS

These form the heart of your script. An action is usually considered to be any line that instructs Flash to do, set, create, change, load, or delete something.

Here are some examples of actions from the sample script:

```
var mugCost:Number = 5.00;  
cashRegister_mc.gotoAndPlay (50);
```

The first line creates a variable named `mugCost`, sets its data type as `Number` (indicating the variable will hold a numeric value), and sets the value of the variable to `5.00`.

The second line tells the **cashRegister_mc** movie clip instance to begin playing at Frame 50 of its timeline.

Generally speaking, most of the lines in a script that are within curly braces (`{}`) are actions. These lines are usually separated by semicolons (we’ll discuss punctuation shortly).

OPERATORS

These include a number of symbols (`=`, `<`, `>`, `+`, `-`, `*`, `&&`, etc.) and are used to connect two elements in a script in various ways. Take a look at these examples:

- `var taxPercent:Number = .06;` assigns a numeric value of `.06` to the variable named `taxPercent`
- `amountA < amountB` asks if `amountA` is less than `amountB`
- `value1 * 500` multiplies `value1` times `500`

KEYWORDS

These are words reserved for specific purposes within ActionScript syntax. As such, they cannot be used as variable, function, or label names. For example, the word `on` is a keyword and can only be used in a script to denote an event that triggers a script, such as `on (press)`, `on (rollOver)`, `on (rollOut)`, and so on. Attempting to use keywords in your scripts for anything other than their intended purpose will result in errors.

Other keywords include `break`, `case`, `class`, `continue`, `default`, `delete`, `do`, `dynamic`,

else, extends, finally, for, function, get, if, implements, import, interface, in, instanceof, new, null, private, public, return, set, static, switch, this, throw, try, typeof, undefined, var, void, while, and with.

DATA

A dynamic script almost always creates, uses, or updates various pieces of data during its execution. Variables are the most common pieces of dynamic data found in scripts and represent pieces of data that have been given unique names. Once a variable has been created and assigned a value, that value can be accessed anywhere in the script simply by inserting the variable's name.

NOTE *Variable names are case sensitive: myVariable and MyVariable are not the same.*

In our sample script, we created a variable named `mugCost` and assigned it a value of 5.00. Later in the script, the name of that variable is used to refer to the value it contains.

CURLY BRACES

Generally, anything between opening and closing curly braces signifies an action or set of actions the script needs to perform when triggered. Think of curly braces as saying, "As a result of this—{do this}." For example:

```
on (release) {
    //set the cost of the mug
    var mugCost:Number = 5.00;
    //set the local sales tax percentage
    var taxPercent:Number = .06;
}
```

SEMICOLONS

Appearing at the end of most lines of scripts, semicolons are used to separate multiple actions that may need to be executed as the result of a single event (similar to the way semicolons are used to separate thoughts in a single sentence). This example denotes six actions, separated by semicolons:

```
var mugCost:Number = 5.00;
var taxPercent:Number = .06;
var totalTax:Number = mugCost * taxPercent;
var totalCost:Number = mugCost + totalTax;
myTextBox_txt.text = "The total cost of your transaction is " + totalCost;
cashRegister_mc.gotoAndPlay (50);
```

DOT SYNTAX

Dots (.) are used within scripts in a couple of ways: One is to denote the target path to a specific timeline. For example, `_root.usa.indiana.bloomington` points to a movie clip on the main (`_root`) timeline named **usa**, which contains a movie clip named **indiana**, which contains a movie clip named **bloomington**.

Because ActionScript is an object-oriented language, most interactive tasks are accomplished by changing a characteristic (*property*) of an object or by telling an object to do something (*invoking a method*). When changing a property or when invoking a method, dots are used to separate the object's name from the property or method being worked with. For example, movie clips are objects; to set the rotation property of a movie clip instance named **wheel_mc**, you would use the syntax:

```
wheel_mc._rotation = 90;
```

Notice how a dot separates the name of the object from the property being set.

To tell the same movie clip instance to play, invoking the `play()` method, you would use the syntax:

```
wheel_mc.play()
```

Once again, a dot separates the name of the object from the method invoked.

PARENTHESES

These are used in various ways in ActionScript. For the most part, scripts employ parentheses to set a specific value that an action will use during its execution. Look at the last line of our sample script that tells the **cashRegister_mc** movie clip instance to go to and play Frame 50:

```
cashRegister_mc.gotoAndPlay (50);
```

If the value within parentheses is changed from 50 to 20, the action still performs the same basic task (moving the **cashRegister_mc** movie clip instance to a specified frame number); it just does so according to the new value. Parentheses are a way of telling an action to work based on what's specified between the parentheses.

QUOTATION MARKS

These are used to denote textual data in the script. Because text is used in the actual creation of the script, quotation marks provide the only means for a script to distinguish between instructions (pieces of data) and actual words. For example, Derek (without quotes) signifies the name of a piece of data. On the other hand, "Derek" signifies the actual word "Derek."

COMMENTS

These are lines in the script preceded by two forward slashes (//). When executing a script, Flash ignores lines containing comments. They indicate descriptive notes about what the script is doing at this point in its execution. Comments enable you to review a script months after it was written and still get a clear idea of its underlying logic.

You can also create multi-line comments using the syntax:

```
/* everything between  
here is considered  
a comment */
```

INDENTING/SPACING

Although not absolutely necessary, it's a good idea to indent and space the syntax in your code. For example:

```
on (release) {  
  var mugCost:Number = 5.00;  
}
```

will execute the same as:

```
on (release) {  
  var mugCost:Number = 5.00;  
}
```

However, by indenting code, you make it easier to read. A good rule is to indent anything within curly braces to indicate that the code within those braces represents a *code block*, or chunk of code, that is to be executed at the same time. (The AutoFormat feature of the Actions panel takes care of most of this for you.) You can *nest* code blocks within other code blocks—a concept that will become clearer as you work through the exercises.

For the most part, white space is ignored within a script. For example:

```
var totalCost:Number = mugCost + totalTax ;
```

will execute in the same way as:

```
var totalCost:Number =mugCost+totalTax;
```

While some programmers feel that extra white space makes their code easier to read, others believe it slows them down to insert spaces. For the most part, the choice is yours. There are a couple of exceptions: variable names cannot contain spaces; nor can you put a space between an object name and an associated property or method. While this syntax is acceptable:

```
myObject.propertyName
```

this is not:

```
myObject. propertyName
```

In addition, there must be a space between the `var` keyword used when creating a variable, and the actual name of the variable. This is correct:

```
var variableName
```

but this is not:

```
varvariableName
```

USING THE ACTIONS PANEL/ACTIONSCRIPT EDITOR

Obviously, the premise of this book requires you to concentrate on writing scripts. It's a good idea to get familiar with the tools you'll use in Flash to do so. In this, your first exercise, you'll learn some of the basics of creating scripts with the ActionScript editor.

NOTE *The purpose of this book is to teach ActionScript, not so much how to use the Flash interface. This discussion will be concise, providing enough information to help you progress through the book. For a more extensive look at the many features of the Actions panel, pick up the Macromedia Flash Visual QuickStart Guide.*

1) With Flash open, choose File > New and choose Flash Document from the list of choices. Press OK.

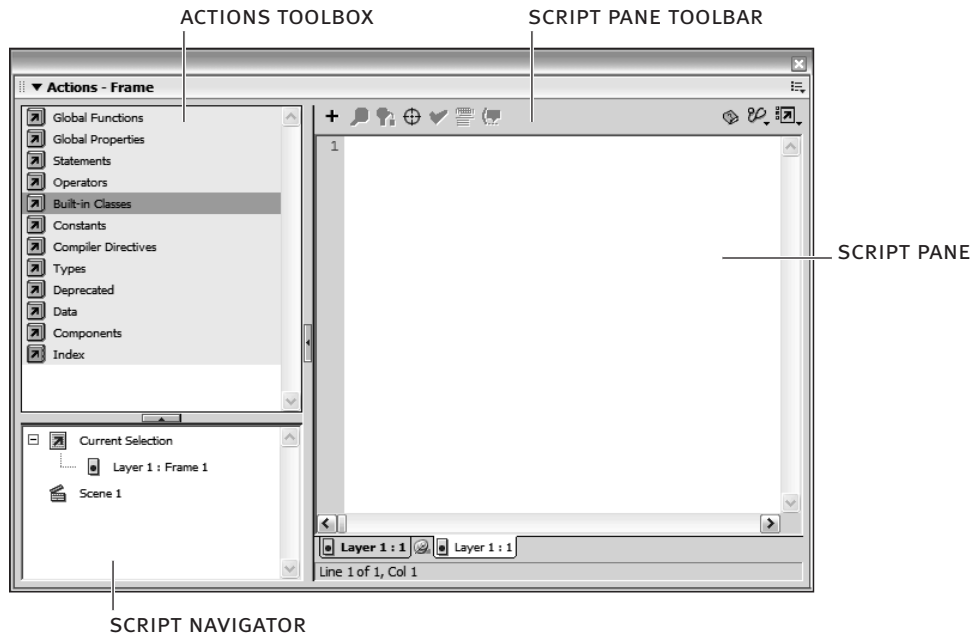
This step creates a new Flash document. It's usually a good idea to give your document a name and save it, so we'll do that next.

2) From the File menu, choose Save As. In the dialog box that appears, navigate to any folder on your hard drive where you would like to save this file (it's not important where, really), name it *myFirstFile fla*, and press the Save button.

After you press Save, the tab at the top of the document window will reflect the name of your new file.

3) Open the Actions panel by choosing Window > Development Panels > Actions.

Let's look at the various sections of the Actions panel.



The *Script pane* is where you add ActionScript. You type into this window just as you would a word processor. The script that appears in this window changes, depending on the currently selected element in the Flash authoring environment. For example, selecting a keyframe on Frame 10 allows you to place a script on that frame if one doesn't already exist; if that frame already contains a script, it will be displayed for editing.

The *Actions toolbox* contains a categorical list of ActionScript elements. Double-clicking an icon (a book with an arrow) opens or closes a category in the list. The toolbox is designed to provide a quick way of adding script elements to your scripts for further configuration. You can add script elements to the Script pane by double-clicking the element's name in the toolbox window, or by clicking and dragging it to the Script pane.

The *Script Navigator* displays a hierarchical list of elements (frames, buttons, movie clip instances) in your projects that contain scripts. Clicking an element will display the script attached to it in the Script pane, allowing you to navigate quickly through the scripts in your project for editing purposes. Only elements with scripts attached to them appear in the Script Navigator.

The *Script pane toolbar* appears above the Script pane and provides a series of buttons and commands, enabling you to add to or edit the current script in the Script pane in various ways.

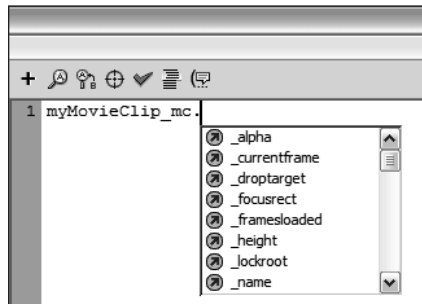
In the next steps, we'll explore the capabilities and functionalities of the ActionScript editor. Let's look first at code hinting.

4) In the Script pane, type:

```
myMovieClip_mc.
```

NOTE *Let's assume this is the name of a movie clip instance we've placed in our movie.*

Immediately after you type the dot (.), a drop-down menu provides a list of actions applicable to movie clip instances.



Why did the editor provide a list of commands for movie clip instances? It's because of the name we chose for our movie clip instance. Notice that we added **_mc** to the movie clip instance's name. This suffix enables the editor to automatically identify **myMovieClip_mc** as a movie clip instance and provide a drop-down list of appropriate commands when you type that name in the Script pane.

Common suffixes for visual movie elements include **_btn** for buttons and **_txt** for text fields. We will be using these suffixes for visual elements throughout this book.

Other, nonvisual elements, such as Sound and Date objects, have suffixes as well, but instead of using suffixes for nonvisual elements, we'll instead be utilizing a new functionality in Flash MX 2004, which we'll demonstrate next.

NOTE *For a complete list of suffixes, consult the ActionScript Dictionary.*

5) Select the current script and delete it. In its place, type:

```
var mySound:Sound = new Sound();
```

This line of script creates a new Sound object named mySound. Creating a Sound object using this syntax identifies mySound as a Sound object to the ActionScript editor. As a result, referencing this object by its name later in the script will cause a drop-down menu of appropriate Sound object–related commands to appear automatically. Let’s test it.

6) Press Enter/Return to go to the next line in the Script pane, and type:

```
mySound.
```

Once again, immediately after typing the dot, you see a drop-down menu with a list of actions applicable to Sound objects.

You can create a new Sound object using this syntax:

```
mySound = new Sound();
```

but this syntax will not activate the functionality of code hinting when you script that object, as the syntax in Step 5 does.

To help you grasp this concept, let’s look at a couple more examples.

This code activates Color object–related code hinting for the myColor Color object:

```
var myColor:Color = new Color();
```

This code does not:

```
myColor = new Color();
```

This code activates Date object–related code hinting for the myDate Date object:

```
var myDate:Date = new Date();
```

This code does not:

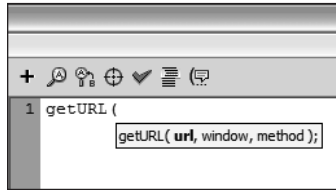
```
myDate = new Date();
```

NOTE *To clarify, in this book we will be using suffixes only when naming visual elements such as movie clip instances, text fields, and buttons. This is because visual elements are not created in the same manner as the description in this step. Using suffixes in their names is the only way to activate code hinting when referencing them in the ActionScript editor.*

Let’s look next at another type of code hint available in the ActionScript editor.

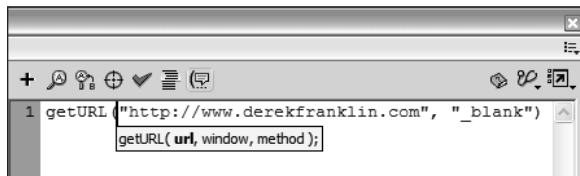
7) Select the current script and delete it. Type in its place:

```
getUrl (
```



Once you type the opening parenthesis, a code hint Tooltip will appear. This functionality becomes active when you type the code for an action that has definable parameters, such as the `getUrl()` action. The currently definable parameter will appear in bold text within the Tooltip. As you enter various parameter data (separated by commas), the Tooltip will be updated to display the currently definable parameter in bold text. The Tooltip will disappear once you type the closing parenthesis for the action.

You can manually display code hints at any time by placing the insertion point (cursor) at a location where code hints normally appear automatically (such as just after the opening parenthesis of an action), then pressing the Code Hint button on the Script pane toolbar.



Other buttons on the toolbar allow you to find words in your scripts, replace words in your scripts (a great feature if you change the name of something), insert target paths, check scripts (without having to spend extra time to actually export the entire movie), and autoformat your scripts (allowing the editor to handle the job of formatting your scripts in an easy-to-read manner).

The last aspect of the ActionScript editor that we'll discuss here is its ability to create **.as** files.

8) From the File menu, choose New and select ActionScript File from the list of choices. Press OK.

This step creates a new ActionScript file, ready for editing in the ActionScript editor.

The ActionScript editor looks very similar to how it looked in the previous steps. The most noticeable difference is the fact that while the interface elements of the editor are active, the rest of Flash’s interface is dimmed out. Flash enters this “limited” mode when you edit **.as** files. The reason is that in this mode its sole purpose is to enable you to create and edit **.as** files. Drawing tools and other panel functionalities have no meaning in this context.

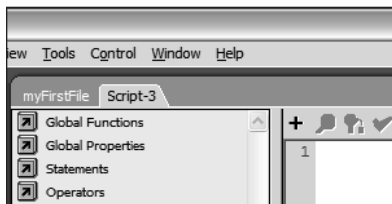
TIP *You can rid the interface completely of dimmed-out interface elements by pressing F4. Pressing F4 again brings them back.*

Another noticeable change is the disappearance of the Script Navigator. Once again, because this mode is meant for editing **.as** files, the concept of navigating various scripts in a project has no meaning, and the Script Navigator is nonexistent in this mode. Other than these differences, the ActionScript editor works in the same manner as previously discussed in this exercise.

When you create **.as** files (as you’ll be doing later in the book), you save them by choosing File > Save and providing a name.

NOTE *In most cases in this book, unless otherwise stated, **.as** files (once we begin creating them) should be saved in the same directory as your main **.fla** files.*

One last thing to note about the Flash authoring environment is that two tabs now appear at the top of the document window. One tab represents the Flash document we created in Step 1, and the other represents the open ActionScript file we just created. Clicking on these tabs allows you to switch easily between authoring the Flash document and the ActionScript file. Flash automatically switches editing modes when you do.



This completes the exercise. Next, you start creating your first application.

PLANNING A PROJECT

When creating a project that contains a generous amount of ActionScript, it's wise to do some planning up front. Dealing with problems in the idea stage makes a lot more sense than dealing with them in the development stage, where they often require more time and cause frustration before they are fixed. We guarantee you'll save time in the long run.

Many issues must be addressed before you even open Flash and start scripting. A good way to go about this is to ask yourself a series of questions.

WHAT DO YOU WANT TO OCCUR?

This is the most important question in the script planning process. Be as clear, informative, and visual as possible in your answer, but avoid going into too much detail.

For the project we discuss in this lesson, we want to create a scene that acts as a front end for paying an electric bill. We want the amount of the bill to be loaded into the movie from an external source, a text file. We want to allow the user to enter an amount to pay into a text box. When a button is pressed, the amount the user paid will be compared to the amount he or she owed, and a visual and textual representation (a custom message) of the result—overpaid, underpaid, or paid in full—will be presented. When the user releases that button, we want the visual and textual elements in the scene to return to their original state. The script that accomplishes this will be the main script in the project.

WHAT PIECES OF DATA DO YOU NEED TO TRACK?

In other words, what numbers or values in the application are integral to its function? In our case, that data is the amount of the electric bill. We will also need to keep track of the difference between what the user owes and what he or she has paid, so that we can display that value in custom messages.

WHAT NEEDS TO HAPPEN IN THE MOVIE PRIOR TO A SCRIPT BEING TRIGGERED?

In our project, the amount of the electric bill must be established in the movie before anything else can happen. Because the primary goal of our project is to compare the amount of the electric bill with the amount the user chooses to pay, if the amount of the electric bill isn't established when the movie first plays, there will be nothing to compare when the script is executed. Creating and setting data prior to a script's being executed, or when a movie first plays, is known as *initializing* the data—a common practice in scripting and something that's usually transparent to the user.

At this point, you need to start thinking about how the data—the amount of the electric bill—will get into the movie. You can place it within the movie when you author it, or you can have it loaded from an external source (for example, a server or text file) when the movie plays. For our project, we opt for the latter: We use a simple script to load a text file containing the amount of the electric bill into the movie. The text file loaded into the movie to provide data is known as a *data source*.

WHAT EVENT WILL TRIGGER THE MAIN SCRIPT?

In our case, the answer is obvious: a button press. However, all kinds of events can trigger a script in Flash, so it's important to give some thought to this question. Does something need to happen when a user moves, presses, or releases the mouse, or when he or she presses a key on the keyboard? How about when a movie clip first appears in the scene? Or does the event need to happen continually (the whole time the movie is playing)? We'll discuss such events in detail in the next lesson.

ARE THERE DECISIONS TO BE MADE WHEN THE MAIN SCRIPT IS TRIGGERED?

When the main script in our movie is triggered, the amount the user enters to pay needs to be compared with the amount he or she owes to determine whether the payment amount is too much, too little, or right on target. The answers to these questions will determine the custom message to display as well as what other visual elements are visible on the screen.

WHAT ELEMENTS MAKE UP THE SCENE? HOW DO THEY FUNCTION?

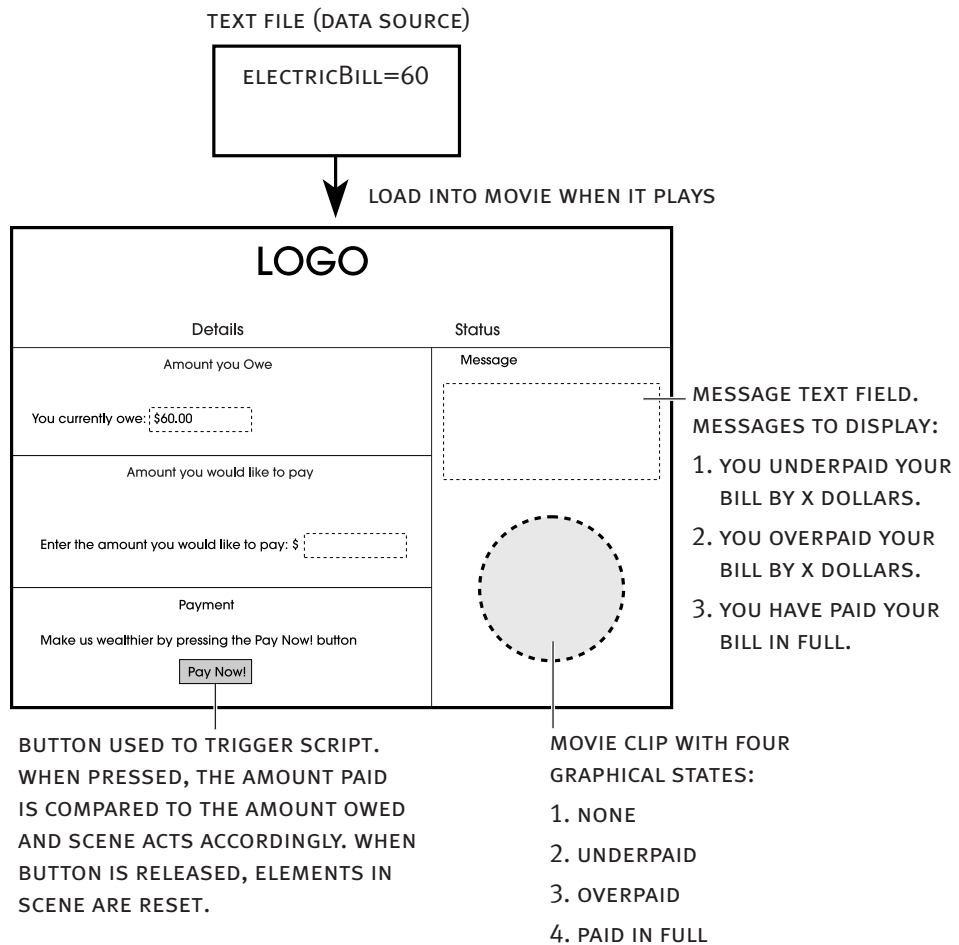
Our scene will be made up of a number of elements, some of which we need to name so that ActionScript can use, control, and/or interact with them. To trigger the script, our scene will need a button, which will look like a pay button found on many sites.

We also need a dynamic text field to display the amount of the bill; we'll name this text field **owed_txt**. In addition, we need an input text field where the user can enter the amount he or she wishes to pay; we'll name this text field **paid_txt**. We also need a dynamic text field to display the custom message generated by the script; we'll name this text field **message_txt**. Finally, we'll add a graphic of a rubber stamp in the form of a movie clip instance with four visual states. Initially, the stamp will not be visible. If the user has not paid enough, a stamp will appear, indicating underpayment. A stamp showing payment in full will appear if the user pays the exact amount, and another showing overpayment if the user overpays. This rubber-stamp movie clip instance will be named **stamp_mc**.

WHAT WILL YOUR SCENE LOOK LIKE?

Use whatever means you want—an illustration program or a pencil and napkin—to create a rough graphical representation of your scene (both its appearance and the action that will take place), as shown in the diagram below. Include all the information you've gathered at this point. This important part of the planning process is often called *storyboarding*.

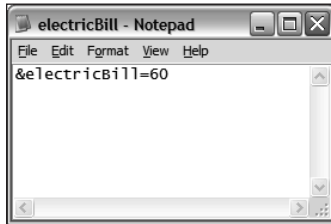
As you grow more proficient at ActionScript and develop additional projects, you'll soon be able to ask (and answer) the previous planning questions intuitively. However, no matter what your skill level, storyboarding remains an essential part of the planning process.



WRITING YOUR FIRST SCRIPT

Now that we have all the information we need, as well as a rough storyboard for our project, let's assemble it.

1) Open Windows Notepad or Apple Simple Text to create a new text file and type `&electricBill=60`.



In the first part of this exercise we create the source for the data to be loaded into our movie when it plays. For our project, this data source contains a value representing the amount of the electric bill. When Flash loads this text file, it interprets this line of text and creates within the movie a piece of data (called a *variable*) named `electricBill` and assigns it a value of 60. The value of this variable will be used in several ways in our project.

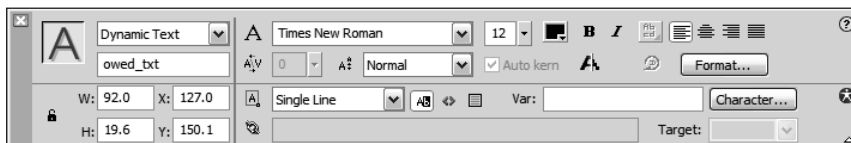
NOTE *Loading data into Flash from external sources is discussed in detail in Lesson 11, "Getting Data In and Out of Flash." Variables are discussed in detail in Lesson 6, "Creating and Manipulating Data."*

2) Name the text file *Electric_Bill.txt* and save it in the folder that contains the files for this lesson. Open *electricbill1 fla* in the Lesson01/Assets folder.

With the exception of setting up scripts, our project elements are largely in place. (Remember, the focus of this book is ActionScript, *not* how to use Flash.)

Our project's main timeline contains five layers. The layers will be named in such a way that their content is evident. Currently, the Actions layer is empty.

3) Open the Property Inspector and select the text field to the right of the text that reads "You currently owe:". In the Property Inspector, give this text field an instance name of *owed_txt*.



Because this text field will be used to display the amount of the electric bill, we've named it **owed_txt**. We add the **_txt** suffix to this text field's name so that when we create the script that references it, code hints will appear.

4) With the Property Inspector open, select the text field to the right of the text that reads "Enter the amount you would like to pay:". In the Property Inspector, give this text field an instance name of *paid_txt*.

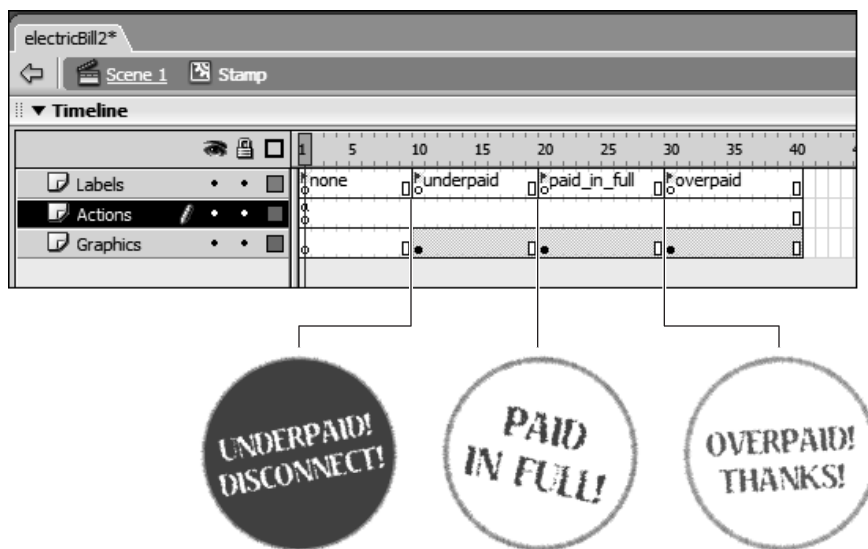
The text field you selected is an input text field. It will be used to accept input from the user—specifically, the amount he or she wants to pay toward the electric bill. We've entered zero (0) in this text field, so that's the amount that will appear initially when our movie plays.

5) With the Property Inspector open, select the text field on the right of the stage, under the label "Status". In the Property Inspector, give this text field an instance name of *message_txt*.

This text field will be used to display a custom message to our user, depending on how much he or she chooses to pay.

6) Select the stamp movie clip instance (it appears as a tiny white circle below the *message_txt* text field) and name it *stamp_mc*.

Because our script will affect this movie clip instance, we must give the movie clip instance a name so that we can tell the script what to do with it. This movie clip instance has four frame labels (*none*, *underpaid*, *paid_in_full*, and *overpaid*) on its timeline that represent how it will look under various conditions. Initially, it will appear as *none*.



In the script we're about to set up, we want the movie clip to move to the *underpaid* label if the user pays less than what he or she owes, which causes the Underpaid stamp graphic to appear. If the user pays the exact amount, we want to send the clip's timeline to the frame label *paid_in_full*, which causes the Paid in Full stamp graphic to appear. If the user pays too much, we want the clip's timeline to go to the frame labeled *overpaid*, causing the Overpaid stamp graphic to appear.

Now that our scene elements are named, we're ready to script. Begin by instructing our movie to load the data from the external text file.

7) With the Actions panel open, select Frame 1 of the Actions layer, and enter the script:

```
var externalData:LoadVars = new LoadVars();
```

ActionScript uses LoadVars objects to load and store data that comes from an external source. The LoadVars object in our example is the external text file named **Electric_Bill.txt** that we created earlier.

In the next steps we add a script that loads the data in our external text file into this object, for use by Flash.

8) Add this script below the current script on Frame 1:

```
externalData.onLoad = function(){  
    owed_txt.text = externalData.electricBill;  
}
```

With these three lines of script, we tell Flash what to do once data from our text file has finished loading into the externalData LoadVars object. We've used the onLoad event to trigger the execution of the rest of the script. Essentially, this script is saying, "When data has finished loading (onLoad) into the externalData LoadVars object, execute the following function." (You'll learn more about functions in Lesson 5, "Using Functions.")

The function does one thing: it sets the value of the text property of the **owed_txt** text field instance to equal the value of externalData.electricBill. This requires a bit of explanation.

Our LoadVars object is named externalData. As mentioned earlier, this object is used to load and store data loaded from an external source. Our text file contains a single piece of data named electricBill, which has a value of 60. When this data is loaded into the externalData LoadVars object, it becomes part of that object. Thus, externalData.electricBill holds a value of 60 once the data from the external text

file has been loaded. If our external text file contained other pieces of data, they could be referenced using the syntax:

```
externalData.name  
externalData.address  
externalData.phone
```

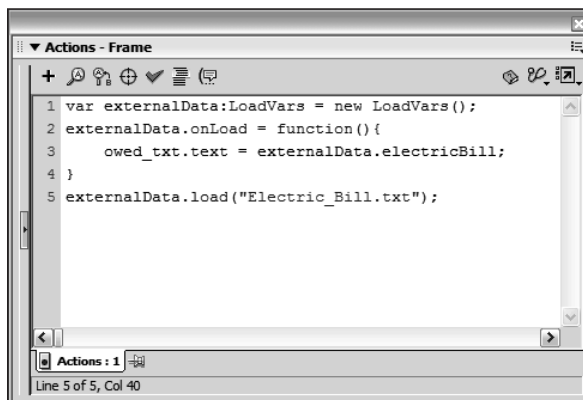
You'll remember that **owed_txt** is the instance name we assigned to the text field used to display what the customer owes. Thus, the function will cause the number 60 to appear in the **owed_txt** text field.

Text fields are objects that have numerous properties. One of the properties of a text field instance is its `text` property. This property is used to designate the text displayed in an instance. The script demonstrates how this property is used to set the text displayed in the text field instance named **owed_txt**. Notice that a dot (.) separates the object name from the name of its property. In ActionScript, this is how you indicate that you want to work with something specific concerning the object—in this case, one of its properties.

Here you'll also notice the use of an operator between two elements—an object property and a variable (and its value). Here, the equals sign (=) is used to tell the script to assign the value of the `externalData.electricBill` variable to the `text` property of the **owed_txt** text field instance.

9) Add this script below the current script on Frame 1:

```
externalData.load("Electric_Bill.txt");
```



This line of script tells Flash to begin loading of the external data. It does so using the `load` command, which is a special command used by `LoadVars` objects. The command requires that you specify the URL of the external data to be loaded. In our case, it's the text file named **Electric_Bill.txt**. This is an example of setting a parameter value.

Recall that earlier we discussed how some actions had configurable parameters, and the parameter value you provided would determine how the action worked. In this case, the load action allows us to enter the URL to a text-based data source anywhere on our hard drive or on the Web. The command always works in the same manner (it loads data from an external source), but the location of the data source (the parameter value) can vary. You'll find that many actions in ActionScript work in this manner.

The script required for loading data from our external text file into Flash is complete. Let's do a quick review.

First, we created a LoadVars object named `externalData` for loading and holding our external data. Next, we told the `externalData` object to execute an action after data had finished loading into it. Finally, we initiated the process of loading the external data.

Are you wondering why we told the script what to do with the loaded data before we loaded the data? While it may not be obvious, it's the logical thing to do. Look at nature. For example, our stomachs are preprogrammed to process food. Imagine the mess that would ensue if we had to eat a complete meal before our stomachs knew what to do with the food. The same principle applies here. Scripting generally requires that you script how to handle various consequences, before they actually occur.

NOTE *We've placed this script on Frame 1 of our movie so that it will be executed as soon as the movie begins playing—important because our movie needs the `electricBill` value in order for the script that we'll be adding to the Pay Now! button to work.*

Now it's time to set up the script for the Pay Now! button.

10) With the Actions panel open, select the Pay Now! button and enter this script:

```
on (press) {  
    var amountPaid:Number = Number(paid_txt.text);  
    var amountOwed:Number = Number(owed_txt.text);  
}
```

This script executes when the button it is attached to is pressed. In the script, this event is followed by an opening curly brace (`{`), a couple lines of script, and a closing curly brace (`}`). These curly braces indicate, “Do these two things when the button is pressed.”

In the first line of the script, a variable named `amountPaid` is created and its value is set to equal that displayed in the **paid_txt** text field instance—with a twist. Normally, anything entered in a text field is considered text. Thus, even if a value of 100 appears in the text field as numerals, it's considered text (consisting of the characters 1, 0, and 0, or "100" with quotes) rather than the number 100 (without quotes).

NOTE *The fact that Flash considers everything entered in a text field to be text is a default behavior. Even though the user might not add quotes while typing into a text field, Flash adds them so that the movie knows that the value is a text value.*

You can think of the `Number()` function as a specialized tool that allows you to convert a text value to a numerical value in such a way that ActionScript recognizes it as a number instead. You need to place the text you want to convert between the parentheses of the function. For example:

```
var myNumber:Number = Number("945");
```

will convert the text **"945"** to the number 945 (no quotes) and assign that number value to the variable named `myNumber`. In our script, we used the `Number()` function and placed a reference to the text value to be converted between the parentheses of the function. If the user types **"54"** (text initially) into the **paid_txt** field, the `Number()` function causes `amountPaid` to be given a numeric value of 54 as well.

TIP *The `Number()` function has its limitations: you can use it only to convert something that is potentially a number in the first place. For example, `Number("dog")` results in `NaN` (not a number) because a numeric conversion is not possible.*

The next line in the script in Step 10 does essentially the same thing, except that the value of `amountOwed` is set to equal the converted-to-a-number value displayed in the **owed_txt** text field instance. You'll remember that this text field instance displays the amount of the electric bill. Thus, after the conversion takes place, `amountOwed` is given a value of 60.

The reason for converting the values in the text fields in this manner is that most of the rest of the script will be using them to make mathematical evaluations or calculations—it needs to see them as numbers, not text, in order for them to work.

In summary, when the button is pressed, the text values displayed in the **paid_txt** and **owed_txt** text fields are converted to numbers. These number values are assigned to the variables named `amountPaid` and `amountOwed`, respectively. These variable values will be used in the remainder of the script.

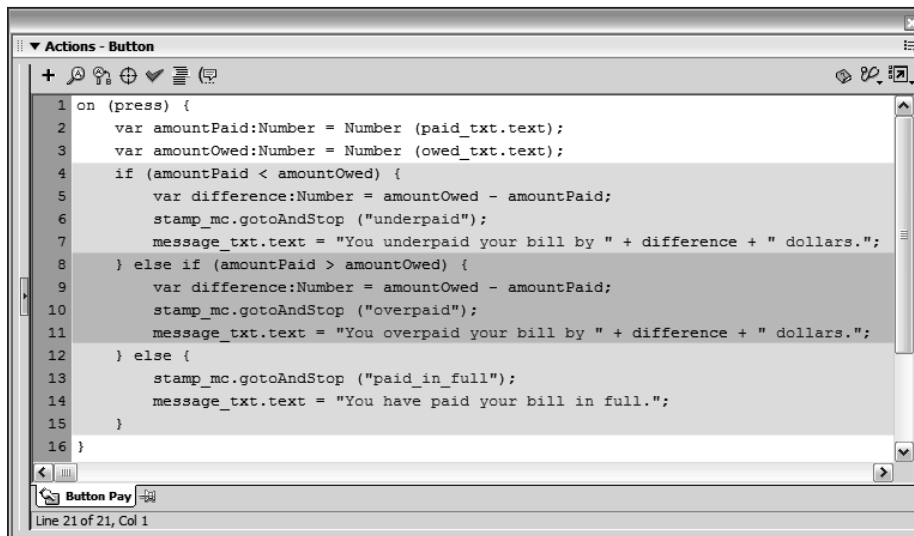
11) With the Actions panel still open, add these lines to the script created in the previous step. Add them within the curly braces ({}), just below where it says `var amountOwed:Number = Number(owed_txt.text);`

```
if (amountPaid < amountOwed) {
    var difference:Number = amountOwed - amountPaid;
    stamp_mc.gotoAndStop ("underpaid");
    message_txt.text = "You underpaid your bill by " + difference + " dollars.";
} else if (amountPaid > amountOwed) {
    var difference:Number = amountOwed - amountPaid;
    stamp_mc.gotoAndStop ("overpaid");
    message_txt.text = "You overpaid your bill by " + difference + " dollars.";
} else {
    stamp_mc.gotoAndStop ("paid_in_full");
    message_txt.text = "You have paid your bill in full.";
}
```

Because we added these lines of script *within* the curly braces of the on (press) event, they are also executed when the button is pressed.

This part of the script is broken into three parts, identified by the lines:

```
if (amountPaid < amountOwed)
else if (amountPaid > amountOwed)
else
```



These three lines represent a series of conditions the script will analyze when executed. The condition to be analyzed in each case is specified between the parentheses. Underneath each of these lines in the script are several lines of indented script (between additional curly braces), which represent the actions that will be executed if that particular condition proves true. Here's how it works:

When our Pay Now! button is pressed, we want our script to determine whether the amount the user enters to pay is more than, less than, or equal to the amount owed. That's what the three conditions our script analyzes are all about. Let's review the first condition, which looks like this:

```
if (amountPaid < amountOwed) {
    var difference:Number = amountOwed - amountPaid;
    stamp_mc.gotoAndStop ("underpaid");
    message_txt.text = "You underpaid your bill by " + difference + " dollars.";
}
```

The first line uses a less-than operator (<) to compare the value of one variable to another. It basically states, “If the amount the user enters to pay (*amountPaid*) is *less* than the amount he or she owes (*amountOwed*), take these actions.” These actions are only executed if this condition is true. And if they are executed, it's as a group, which is why they're placed within their own set of curly braces. The first action creates a variable named *difference* and assigns it a value of *amountOwed* minus *amountPaid*. If the user has entered 40 as the amount to pay, *difference* would have a value of 20 (*amountOwed* – *amountPaid*, or 60 – 40). It's important to note that any equation to the right of the equals sign is calculated prior to assigning the calculated value to the item to the left. The next line tells the **stamp_mc** movie clip instance to go to and stop at the frame labeled *underpaid*, causing the Underpaid stamp to appear. The last line will generate a custom message to be displayed in the **message_txt** text field.

We call this a *custom* message because of the way the message is constructed within the script. Note the use of the *difference* variable in this line: the value of *difference* is inserted in the middle of this message between the two sections of quoted text and the plus signs. If *difference* has a value of 20, this message would read: “You underpaid your bill by 20 dollars.”

Remember that anything within quotes is considered plain text. Because *difference* is not enclosed in quotes, ActionScript knows that it references a variable's name and thus will insert that variable's value there. The plus sign (+) is used to concatenate (join) everything to create a single message. The equals (=) sign is used to assign the final, concatenated value to the text property of the **message_txt** text field.

If the amount the user enters to pay is the same as what's owed, or more, this part of the script is ignored and the next part of the script is analyzed. It reads:

```
else if (amountPaid > amountOwed) {
    var difference:Number = amountOwed - amountPaid;
    stamp_mc.gotoAndStop ("overpaid");
    message_txt.text = "You overpaid your bill by " + difference + " dollars.";
}
```

The first line here states, “If the amount the user enters to pay is more than the amount he or she owes, take these actions.” The actions executed here are variations

on the ones discussed previously, with minor differences. The first difference is that the **stamp_mc** movie clip instance is sent to the frame labeled *overpaid*, which displays the Overpaid stamp. The second difference comes in the wording of the custom message. Instead of saying **underpaid**, here it says **overpaid**. The actions in this section are executed only if the user pays more than what's owed. If that's not the case, these actions are ignored and the last part of the script is analyzed. It reads:

```
else {
    stamp_mc.gotoAndStop ("paid_in_full");
    message_txt.text = "You have paid your bill in full.";
}
```

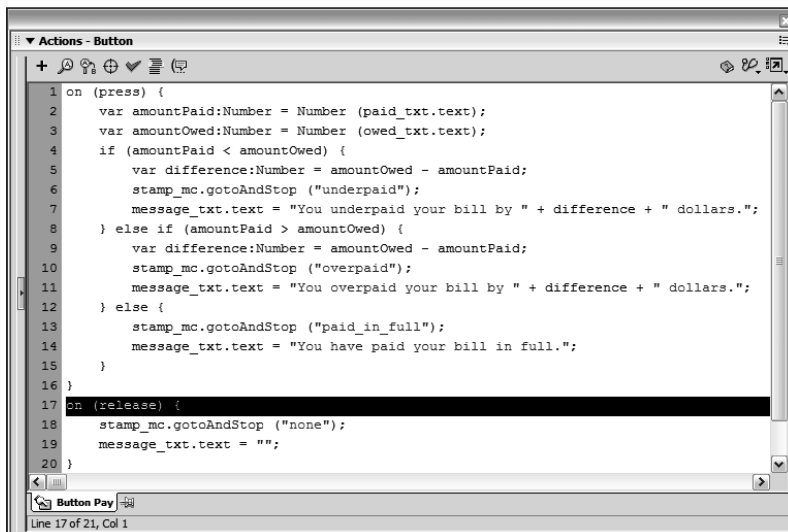
Here, the script doesn't begin by asking if amountPaid is more or less than amountOwed (as it did in the first two sections). This is because the script continues to this point only if the user has entered the exact amount owed—this part of the script will execute only if neither of the first two sections does.

In the end, when the button is pressed, only one of these three sets of actions will execute. As a result, the **stamp_mc** movie clip instance will appear a certain way and a custom message will appear in the **message_txt** text field.

When the Pay Now! button is released, we want the elements in our scene to reset so that they appear as they did when the movie first played. Let's add that functionality.

12) With the Actions panel open and the Pay Now! button selected, enter these lines at the end of the current script:

```
on (release) {
    stamp_mc.gotoAndStop ("none");
    message_txt.text = "";
}
```



This script is triggered when the button is released. It will send the **stamp_mc** movie clip instance to the frame labeled *none* and empty the **message_txt** text field, returning these scene elements to their original state.

13) Save this file as *electricBill2.fla* in the same directory where you saved the *Electric_Bill.txt* file created earlier.

We'll use this file in the next exercise in this lesson.

TESTING YOUR FIRST SCRIPT

It would be pure fantasy to believe ActionScripts always work exactly as planned. Just as it's easy to forget a comma or to misspell a word when writing a letter, it's easy to make mistakes when writing scripts—regardless of how familiar you are with ActionScript. However, unlike your letter recipients, Flash is unforgiving when it comes to script errors. In scripting, errors mean bugs, and bugs mean your script either won't work at all or won't work as planned. Luckily, Flash provides some handy ways to test scripts and stamp out bugs.

1) If *electricBill2.fla* isn't already open, open it now.

This is the file we set up in the last exercise. In this exercise, we'll test the project's functionality from within Flash's testing environment.

2) From Flash's menu bar, choose Control > Test Movie.

This command creates a fully functional version of your exported movie and displays it in Flash's testing environment. Although there are all kinds of ways you can test your movie in this environment (determining overall file size, streaming capability, and appearance), we're interested in testing its interactive features—which means doing everything we can to mess it up.

TIP *Enlist as many friends and colleagues as possible to help you test your project. This way you have a greater chance of testing every possible scenario and thus finding all the potential bugs.*

3) Enter various amounts in the "Enter the amount you would like to pay:" text field, then press the Pay Now! button.

- *Enter an amount less than 60.* If you enter 35 in this text field, the message, "You underpaid your bill by 25 dollars" should appear and the **stamp_mc** movie clip instance should show the Underpaid stamp.

- *Enter an amount more than 60.* If you enter 98 in this text field, the message, “You have overpaid your bill by 38 dollars” should appear and the **stamp_mc** movie clip instance should show the Overpaid stamp. This is what should happen. What really happens is that the message shows an overpayment of –38 dollars, not 38 dollars. We log this bug as an error and continue testing.
- *Enter the exact amount of 60.* If you enter 60 into this text field, the message, “You have paid your bill in full” should appear and the **stamp_mc** movie clip instance should show the Paid in Full stamp.
- *Erase everything in this field.* If you do this and press the Pay Now! button, you get a message stating, “You have paid your bill in full” and the **stamp_mc** movie clip instance will show that you paid your bill in full. Obviously, this is wrong; we log this as an error and continue testing.
- *Enter some text.* If you enter anything beginning with a letter and press the Pay Now! button, you get the message, “You have paid your bill in full” and the **stamp_mc** movie clip instance will show the Paid in Full stamp—another obvious mistake, which we log as an error.

TIP *When you find bugs while testing a complex project, sometimes it’s best to stop testing and begin the bug-stomping process immediately (as opposed to logging several bugs first, then attempting to fix them all at once). The reason is that when attempting to eliminate a bug, you may unwittingly introduce a new one. Fixing several bugs at once could result in several new bugs—obviously not what you want. By fixing bugs one at a time, you can better concentrate your bug-squashing efforts and avoid a lot of needless backtracking.*

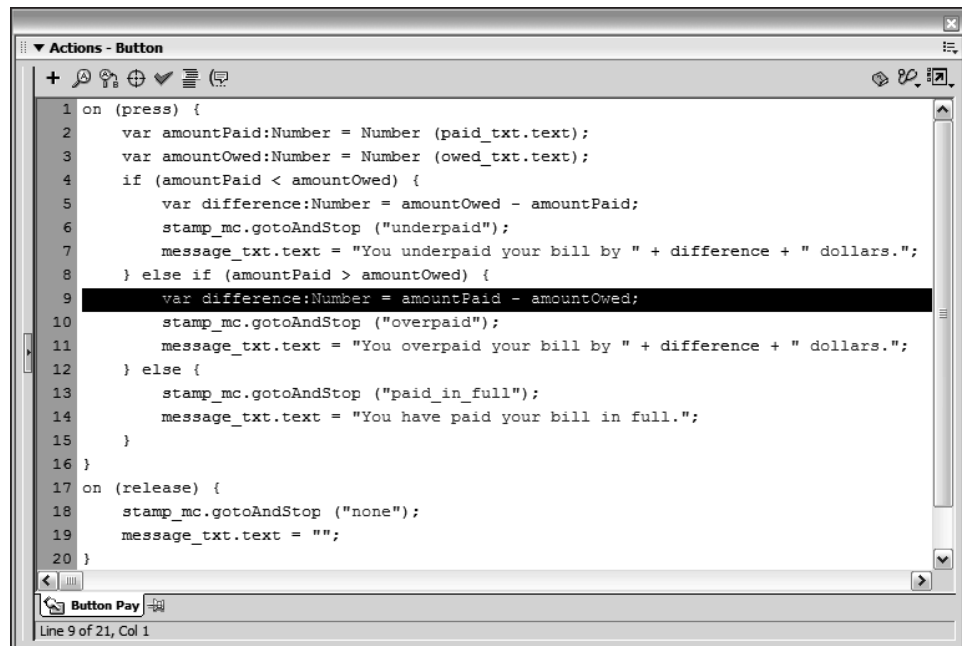
As you know, our project contains three bugs:

- If a user overpays his or her bill, the overage is shown as a negative number in the custom message that is displayed.
- If a user chooses not to pay anything, our project functions incorrectly.
- If a user enters text rather than a numeric value, our project functions incorrectly.

Let’s consider why these bugs occur. In the case of the first bug, we know that the numeric value that appears in this dynamic message is based on the value of the difference variable that’s created when the script is executed. In addition, we know that the problem occurs only if the user pays *more* than the amount of the bill. Thus, the problem lies in the way difference is calculated when the user overpays his or her bill. We’ll review that part of our script.

As for the other two bugs, our script is set up to act in various ways when executed, depending on what amount the user enters to pay. However, we forgot to account for the possibility that the user might not enter anything, or that he or she might enter text, both of which cause our project to act funny. We'll make a slight addition to the script to account for this possibility.

4) Close the testing environment and return to the authoring environment. Select the Pay Now! button and modify Line 9 of the script, which currently reads: `var difference:Number = amountOwed - amountPaid;` to read `var difference:Number = amountPaid - amountOwed;`.



```
1 on (press) {
2   var amountPaid:Number = Number (paid_txt.text);
3   var amountOwed:Number = Number (owed_txt.text);
4   if (amountPaid < amountOwed) {
5     var difference:Number = amountOwed - amountPaid;
6     stamp_mc.gotoAndStop ("underpaid");
7     message_txt.text = "You underpaid your bill by " + difference + " dollars.";
8   } else if (amountPaid > amountOwed) {
9     var difference:Number = amountPaid - amountOwed;
10    stamp_mc.gotoAndStop ("overpaid");
11    message_txt.text = "You overpaid your bill by " + difference + " dollars.";
12  } else {
13    stamp_mc.gotoAndStop ("paid_in_full");
14    message_txt.text = "You have paid your bill in full.";
15  }
16 }
17 on (release) {
18   stamp_mc.gotoAndStop ("none");
19   message_txt.text = "";
20 }
```

In reviewing the section of the script that determines what happens when `amountPaid` exceeds `amountOwed`, we discover that `difference` is calculated by subtracting `amountOwed` by `amountPaid`. How is this a problem? If the user pays 84 dollars, the difference being calculated is $60 - 84$ (or `amountOwed` minus `amountPaid`). Subtracting a larger number from a smaller number results in a negative number. To fix the problem, we simply switch the position of `amountOwed` and `amountPaid` in the line of script that sets the value of `difference`. Now, the smaller number is subtracted from the larger one, resulting in a positive number.

NOTE *You don't need to modify the other area in the script where the value of difference is set, because that area is only executed when the user pays less than he or she owes, in which case the value will be calculated properly.*

5) With the Pay Now! button selected and the Actions panels still open, make this addition and modification to the if statement:

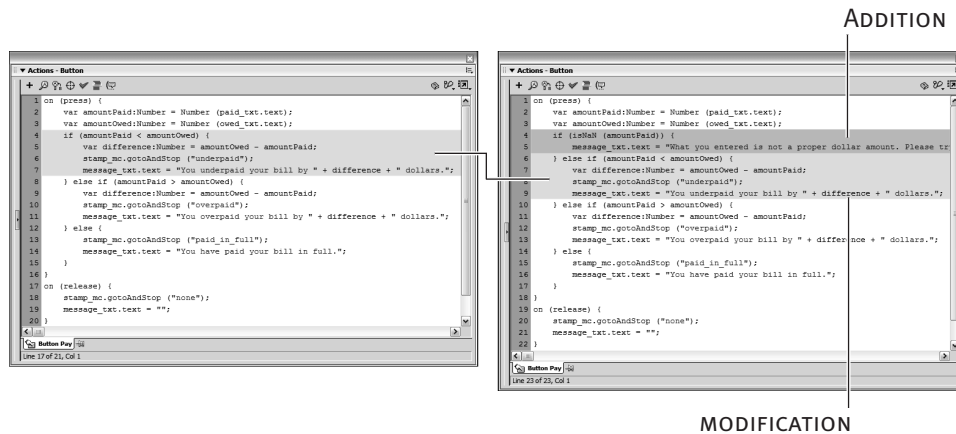
Addition:

```
if (isNaN (amountPaid)) {  
    message_txt.text = "What you entered is not a proper dollar amount. Please try  
    again.";  
}
```

Modification to what used to be the initial if statement:

```
} else if (amountPaid < amountOwed) {  
    var difference:Number = amountOwed - amountPaid;  
    stamp_mc.gotoAndStop ("underpaid");  
    message_txt.text = "You underpaid your bill by " + difference + " dollars.";  
}
```

Notice that with this addition and modification, what used to be the first condition that was analyzed has now been relegated to the second condition. Modifying the script in this way will cause this new condition to be analyzed first.



This addition allows the script to deal with the contingency that the user enters nothing or enters text as the amount to pay. It says that when the Pay Now! button is pressed, if the value of amountPaid is not a number (or isNaN), do nothing in the scene but display a message that asks the user to enter a proper dollar amount.

If the amount entered cannot be converted to a numerical value (for example, “frog”) or if the field is left blank, this part of the script executes. The `isNaN()` function is another special tool that ActionScript provides to take care of simple yet critical tasks. Notice that instead of inserting a literal value between the parentheses of this function (such as “cat” or 57), we’ve placed a reference to a variable’s name (in this case, `amountPaid`). This causes the value that the variable holds to be analyzed.

NOTE *The `isNaN()` function and its uses are covered in greater detail throughout the book.*

We made this the first condition to look for because it’s the most logical thing to check when the script is first executed. If the user enters a numeric value, this part of the script is ignored and the rest of the script works as expected.

6) From Flash’s menu bar, choose Control > Test Movie. In the exported test file, enter various amounts in the "Enter the amount you would like to pay:" text field and press the Pay Now! button.

At this point the movie should work properly under all circumstances.

7) Close the testing environment and return to the authoring environment. Save this file as *electricBill3 fla*.

Congratulations! You’ve completed your first lesson.

WHAT YOU HAVE LEARNED

In this lesson, you have:

- Been introduced to ActionScript 2.0, and learned the differences and similarities between ActionScript 2.0 and ActionScript 1.0 (pages 6–15)
- Familiarized yourself with the various elements that make up a script (pages 15–20)
- Learned to use the ActionScript editor (pages 20–25)
- Planned and developed an ActionScript project (pages 26–28)
- Written the project script (pages 29–38)
- Tested the script, searching for bugs (pages 38–39)
- Fixed bugs to complete the project (pages 39–42)