,ch01.12472 Page 3 Thursday, April 19, 2001 12:58 PM

1

A Gentle Introduction for Non-Programmers

I'm going to teach you to talk to Flash.

Not just to program in Flash but to say things to it and to listen to what it has to say in return. This is not a metaphor or simply a rhetorical device. It's a philosophical approach to programming.

Programming languages are used to send information to and receive information from computers. They are collections of vocabulary and grammar used to communicate, just like human languages. Using a programming language, we tell a computer what to do or ask it for information. It listens, tries to perform the requested actions, and gives responses. So while you may think you are reading this book in order to "learn to program," you are actually learning to communicate with Flash. But, of course, Flash doesn't speak English, French, German, or Cantonese. Flash's native language is ActionScript, and you're going to learn to speak it.

Learning to speak a computer language is sometimes considered synonymous with learning to program. But there is more to programming than learning a language's syntax. What would it be like if Flash could speak English—if we didn't need to learn ActionScript in order to communicate with it?

What would happen if we were to say, "Flash, make a ball bounce around the screen?"

Flash couldn't fulfill our request because it doesn't understand the word "ball." Okay, okay, that's just a matter of semantics. What Flash expects us to describe is the objects in the world it knows: movie clips, buttons, frames, and so on. So, let's rephrase our request in terms that Flash recognizes and see what happens: "Flash, make the movie clip named `ball_one` bounce around the screen."

,ch01.12472 Page 4 Thursday, April 19, 2001 12:58 PM

Flash still can't fulfill our request without more information. How big should the ball be? Where should it be placed? In which direction should it begin traveling? How fast should it go? Around which part of the screen should it bounce? For how long? In two dimensions or three? Hmm . . . we weren't expecting all these questions. In reality, Flash doesn't ask us these questions. Instead, when Flash can't understand us, it just doesn't do what we want it to, or it yields an error message. For now, we'll pretend Flash asked us for more explicit instructions, and reformulate our request as a series of steps:

1. A ball is a circular movie clip symbol named `ball`.
2. A square is a four-sided movie clip symbol named `square`.
3. Make a new green ball 50 pixels in diameter.
4. Call the new ball `ball_one`.
5. Make a new black square 300 pixels wide and place it in the middle of the Stage.
6. Place `ball_one` somewhere on top of the square.
7. Move `ball_one` in a random direction at 75 pixels per second.
8. If `ball_one` hits one of the sides of the square, make it bounce (reverse course).
9. Continue until I tell you to stop.

Even though we gave our instructions in English, we still had to work through all the logic that governs our bouncing ball in order for Flash to understand us. Obviously, there's more to programming than merely the syntax of programming languages. Just as in English, knowing lots of words doesn't necessarily mean you're a great communicator.

Our hypothetical English-speaking-Flash example exposes four important aspects of programming:

- No matter what the language, the art of programming lies in the formulation of logical steps.
- Before you try to say something in a computer language, it usually helps to say it in English.
- A conversation in one language translated into a different language is still made up of the same basic statements.
- Computers aren't very good at making assumptions. They also have a very limited vocabulary.

Most programming has nothing to do with writing code. Before you write even a single line of ActionScript, think through exactly what you want to do and write

,ch01.12472 Page 5 Thursday, April 19, 2001 12:58 PM

out your system's functionality as a flowchart or a blueprint. Once your program has been described sufficiently at the conceptual level, you can translate it into ActionScript.

In programming—as in love, politics, and business—effective communication is the key to success. For Flash to understand your ActionScript, you have to get your syntax absolutely correct down to the last quote, equal sign, and semicolon. And to assure that Flash knows what you're talking about, you must refer only to the world it knows using terms it recognizes. What may be obvious to you is not obvious to a computer. Think of programming a computer like talking to a child: take nothing for granted, be explicit in every detail, and list every step that's necessary to complete a task. But remember that, unlike children, Flash will do precisely what you tell it to do and nothing that you don't tell it to do.

Some Basic Phrases

On the first day of any language school you'd expect to learn a few basic phrases ("Good day," "How are you," etc.). Even if you're just memorizing a phrase and don't know what each word means, you can learn the effect of the phrase and can repeat it to produce that effect. Once you've learned the rules of grammar, expanded your vocabulary, and used the words from your memorized phrases in multiple contexts, you can understand your early phrases in a richer way. The rest of this chapter will be much like that first day of language school—you'll see bits and pieces of code, and you'll be introduced to some fundamental programming grammar. The rest of the book will build on that foundation. You may want to come back to this chapter when you've finished the book to see just how far you've traveled.

Creating Code

For our first exercise, we'll learn how to add four simple lines of code to a Flash movie. Nearly all ActionScript programming takes place in the Actions panel. Any instructions we add to the Actions panel are carried out by Flash when our movie plays. Open the Actions panel now by following these steps:

1. Launch Flash with a new blank document.
2. On the main timeline, select frame 1 of layer 1.
3. Select Window → Actions.

The Actions panel is divided into two sections: the Script pane (on the right) and the Toolbox pane (on the left). The Script pane houses all our code. The Toolbox pane provides us with quick access to the Actions, Operators, Functions, Properties, and

,ch01.12472 Page 6 Thursday, April 19, 2001 12:58 PM

Objects of ActionScript. You'll likely recognize the Basic Actions, shown in Figure 1-1, from prior Flash versions.

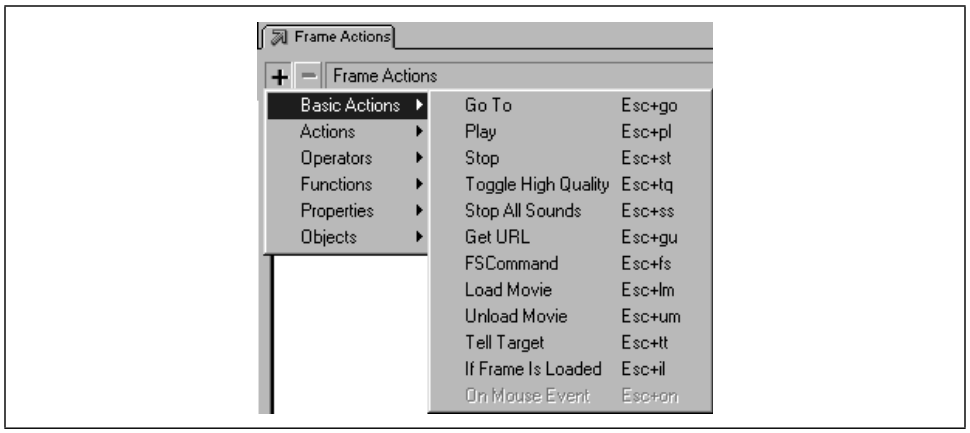

Figure 1-1. Flash 5 Basic Actions

But there's lots more to discover in the Toolbox pane: Figure 1-2 shows all available Actions, including some old friends from Flash 2, 3, and 4. If you continue exploring the Toolbox pane, you'll even find things like Sound, Array, and XML. By the end of this book, we'll have covered them all.

The Toolbox pane's menus may be used to create ActionScript code. However, in order to learn the syntax, principles, and structural makeup of ActionScript, we'll be typing all our code.

So-called *Actions* are more than just Actions—they include various fundamental programming-language tools: variables, conditionals, loops, comments, function calls, and so forth. Although these are lumped together in one menu, the generic name *Action* obscures the programming structures' significance.

We'll be breaking Actions down to give you a programmer's perspective on those structures. Throughout the book, I use the appropriate programming term to describe the Action at hand. For example, instead of writing, "Add a *while* Action," I'll write, "Create a *while* loop." Instead of writing, "Add an *if* Action," I'll write, "Make a new conditional." Instead of writing, "Add a *play* Action," I'll write, "Invoke the *play()* function (or method)." These distinctions are an important part of learning to speak ActionScript.

Ready to get your hands dirty? Let's say hello to Flash.

,ch01.12472 Page 7 Thursday, April 19, 2001 12:58 PM

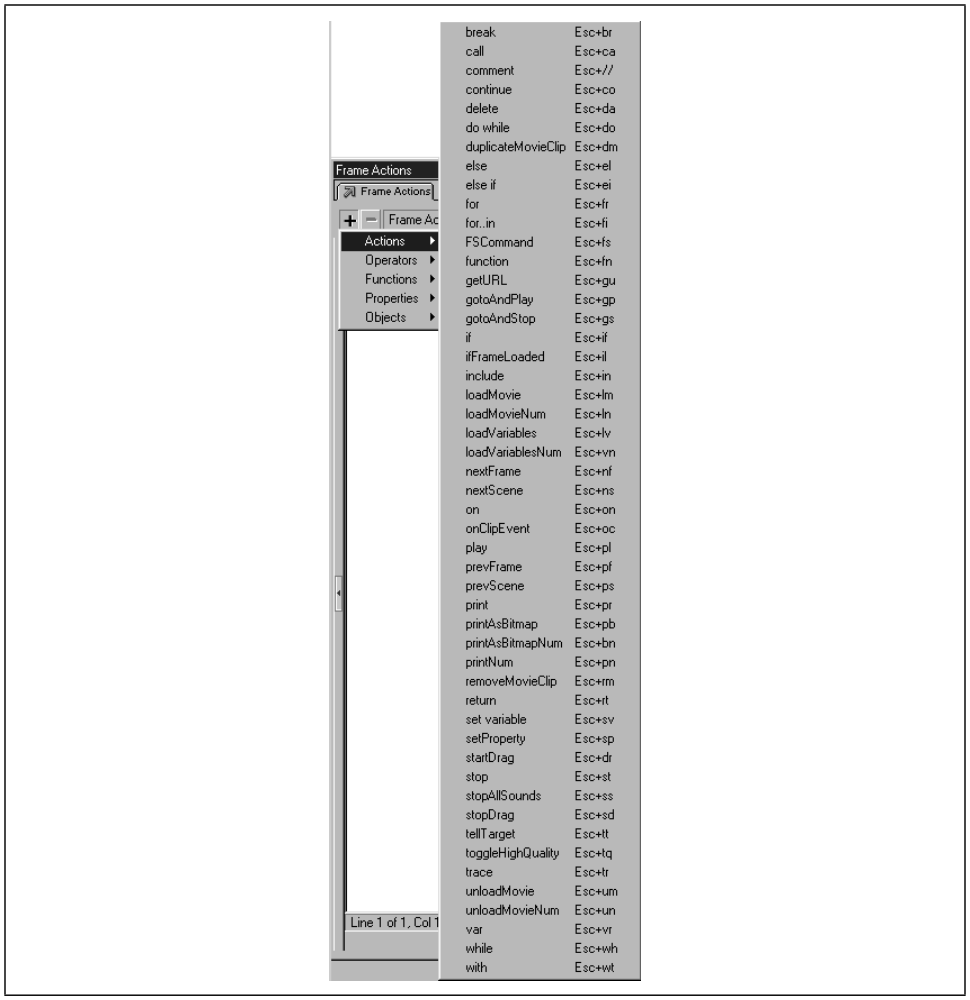

Figure 1-2. Expanded Actions

Say Hi to Flash

Before you can type code into the Actions panel, you must disengage the ActionScript autopilot as follows:

1. Select Edit → Preferences.
2. On the General tab, select Actions Panel → Mode → Expert Mode.
3. Expert Mode is also selectable from the pop-up menu accessible via the arrow at the far right of the Actions panel, though this only sets the current frame's mode. See Chapter 16, *ActionScript Authoring Environment*.

,ch01.12472 Page 8 Thursday, April 19, 2001 12:58 PM

Howdya like that? You're already an expert. When you enter Expert Mode, the Parameters pane at the bottom of the Actions Panel disappears. Don't worry—we're not programming with menus so we won't be needing it.

Next, select frame 1 of layer 1. Your ActionScript (a.k.a., *code*) must always be attached to a frame, movie clip, or button; selecting frame 1 causes subsequently created code to be attached to that frame. In Expert Mode, you can type directly into the Script pane on the right side of the Actions panel, which is where we'll be doing all our programming.

And now, the exciting moment—your first line of code. It's time to introduce yourself to Flash! Type the following into the Script pane:

```
var message = "Hi there, Flash!";
```

That line of code constitutes a complete instruction, known as a *statement*. On the line below it, type your second and third lines of code, shown following this paragraph. Replace *`your name here`* with your first name (whenever you see *`italicized code`* in this book it means you have to replace that portion of the code with your own content):

```
var firstName = "your name here";
trace (message);
```

Hmmm. Nothing has happened yet. That's because our code doesn't do anything until we export a *.swf* file and play our movie. Before we do that, let's ask Flash to say hi back to us. Type your fourth line of code under the lines you've already typed (man, we're really on a roll now . . .):

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

Okay, Flash is ready to meet you. Select Control → Test Movie and see what happens. Some text should appear in the Output window as shown in Figure 1-3.

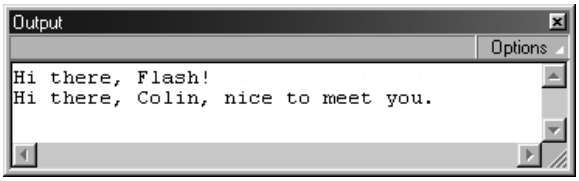

Figure 1-3. Flash gets friendly

Pretty neat, eh?! Let's find out how it all happened.

,ch01.12472 Page 9 Thursday, April 19, 2001 12:58 PM

Keeping Track of Things (Variables)

Remember how I said programming was really just communicating with a computer? Well it is, but perhaps with a little less personality than I've been portraying so far. In your first line of code:

```
var message = "Hi there, Flash!";
```

you didn't really say hi to Flash. You said something more like this:

> Flash, please remember a piece of information for me—specifically, the phrase "Hi there, Flash!" I may need that information in the future, so please give it a label called `message`. If I ask you for `message` later, give me back the text "Hi there, Flash!"

Perhaps not as friendly as saying hi, but it illustrates one of the true foundations of programming: Flash can remember something for you, provided that you label it so that it can be found later. For example, in your second line of code, we had Flash remember your first name, and we named the reference to it `firstName`. Flash remembered your name and displayed it in the Output window when you tested your movie.

The fact that Flash can remember things for us is crucial in programming. Flash can remember any type of data, including text (such as your name), numbers (such as 3.14159), and more complex datatypes that we'll discuss later.

Official variable nomenclature

Time for a few formal terms to describe how Flash remembers things. So far you know that Flash remembers data. An individual piece of data is known as a *datum*. A datum (e.g., "Hi there, Flash!") and the label that identifies it (e.g., `message`) are together known as a *variable*. A variable's label is called its *name*, and a variable's datum is called its *value*. We say that the variable *stores* or *contains* its value. Note that "Hi there, Flash!" is surrounded by double quotation marks (quotes) to indicate that it is a *string* of text, not a number or some other *datatype*.

In your first line of code, you specified the value of the variable `message`. The act of specifying the value of a variable is known as *assigning the variable's value*, or generally, *assignment*. But before you can assign a value to a variable, you must first create it. We formally bring variables into existence by *declaring* them using the special keyword *var*, which you used earlier.

So, in practice, here's how I might use more formal terms to instruct you to create the first line of code you created earlier: Declare a new variable named `message`, and assign it the initial value "Hi there, Flash!" Then you should write:

```
var message = "Hi there, Flash!";
```

,ch01.12472 Page 10 Thursday, April 19, 2001 12:58 PM

The Wizard Behind the Curtain (the Interpreter)

Recall your first two lines of code:

```
var message = "Hi there, Flash!";
var firstName = "your name here";
```

In each of those statements, you created a variable and assigned a value to it. Your third and fourth lines, however, are a little different:

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

These statements use the *trace()* command. You've already seen the effect of that command—it caused Flash to display your text in the Output window. In the third line, Flash displayed the value of the variable `message`. In the last line, Flash also converted the variable `firstName` to its value (whatever you typed) and stuck that into the sentence after the words "Hi there." The *trace()* command, then, causes any specified data to appear in the Output window (which makes it handy for determining what's going on when a program is running).

The question is, what made the *trace()* command place your text in the Output window? When you create a variable or issue a command, you're actually addressing the *ActionScript interpreter*, which runs your programs, manages your code, listens for instructions, performs any ActionScript commands, executes your statements, stores your data, sends you information, calculates values, and even starts up the basic programming environment when a movie is loaded into the Flash Player.

The interpreter translates your ActionScript into a language that the computer understands and uses to carry out your code. During movie playback, the interpreter is always active, dutifully attempting to understand commands you give it. If the interpreter can understand your commands, it sends them to the computer's processor for execution. If a command generates a result, the interpreter provides that response to you. If the interpreter can't understand the command, it sends you an error message. The interpreter, hence, acts like ActionScript's switchboard operator—it's the audience you're addressing in your code and the ambassador that reports back to you from Flash.

Let's take a closer look at how the interpreter works by examining how it handles a simple *trace()* action.

Consider this command as the interpreter would:

```
trace ("Nice night to learn ActionScript.");
```

The interpreter immediately recognizes the keyword *trace* from its special list of legal command names. The interpreter also knows that *trace()* is used to display

,ch01.12472 Page 11 Thursday, April 19, 2001 12:58 PM

text in the Output window, so it also expects to be told which text to display. It finds "Nice night to learn ActionScript." between parentheses following the word *trace* and thinks "Aha! That's just what I need. I'll have that sent to the Output window right away!"

Note that the command is terminated by a semicolon (;). The semicolon acts like the period at the end of a sentence; with few exceptions, every ActionScript statement should end with a semicolon. With the statement successfully understood and all the required information in hand, the interpreter translates the command for the processor to execute, causing our text to appear in the Output window.

That's a gross oversimplification of the internal details of how a computer processor and an interpreter work, but it illustrates these points:

- The interpreter is always listening for your instructions.
- The interpreter has to read your code, letter by letter, and try to understand it. This is the same as you trying to read and understand a sentence in a book.
- The interpreter reads your ActionScript using strict rules—if the parentheses in our *trace()* statement were missing, for example, the interpreter wouldn't be able to understand what's going on, so the command would fail.

You've only just been introduced to the interpreter, but you'll be as intimate with it as you are with a lover before too long: lots of fights, lots of yelling—"Why aren't you listening to me?!"—and lots of beautiful moments when you understand each other perfectly. Strangely enough, my dad always told me the best way to learn a new language is to find a lover that speaks it. May I, therefore, be the first to wish you all the best in your new relationship with the ActionScript interpreter. From now on I'll regularly refer to "the interpreter" instead of "Flash" when describing how ActionScript instructions are carried out.

Extra Info Required (Arguments)

You've already seen one case in which we provided the interpreter with the text to display when issuing a *trace()* command. This approach is common; we'll often issue a command and then provide the interpreter with ancillary data used to execute that command. There's a special name for a datum sent to a command: an *argument*, or synonymously, a *parameter*. To supply an argument to a command, enclose the argument in parentheses, like this:

```
command(argument);
```

When supplying multiple arguments to a command, separate them with commas, like this:

```
command(argument1, argument2, argument3);
```

,ch01.12472 Page 12 Thursday, April 19, 2001 12:58 PM

Supplying an argument to a command is known as *passing* the argument. For example, in the code `gotoAndPlay(5)`, *gotoAndPlay* is the name of the command, and 5 is the argument being passed (in this case the frame number). Some commands, such as *stop()*, require parentheses but do not accept arguments. We'll learn why in Chapter 9, *Functions*.

ActionScript's Glue (Operators)

Let's take another look at your fourth line of code, which contains this *trace()* statement:

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

See the + (plus) signs? They're used to join (*concatenate*) our text together and are but one of many available *operators*. The operators of a programming language are akin to conjunctions ("and," "or," "but," etc.) in human languages. They're devices used to combine and manipulate phrases of code. In the *trace()* example, the plus operator joins the quoted text "Hi there, " to the text contained in the variable `firstName`.

All operators link phrases of code together, manipulating those phrases in the process. Whether the phrases are text, numbers, or some other datatype, an operator nearly always performs some kind of transformation. Very commonly, operators combine two things together, as the plus operator does. But other operators compare values, assign values, facilitate logical decisions, determine datatypes, create new objects, and provide various other handy services.

When used with two numeric operands, the plus sign (+) and the minus sign (–), perform basic arithmetic. The following displays "3" in the Output window:

```
trace(5 - 2);
```

The *less-than* operator checks which of two numbers is smaller or determines which of two letters is alphabetically first:

```
if (3 < 300) {
  // Do something...
}

if ("a" < "z") {
  // Do something else...
}
```

The combinations, comparisons, assignments, or other manipulations performed by operators are known as *operations*. Arithmetic operations are the easiest operations to understand because they follow basic mathematics: addition (+), subtraction (–), multiplication (*), and division (/). But some operators will be less recognizable to you because they perform specialized programming tasks. Take

,ch01.12472 Page 13 Thursday, April 19, 2001 12:58 PM

the *typeof* operator, for example. It tells us what kind of data is stored in a variable. So, if we create a variable `x`, and give it the value 4, we can then ask the interpreter what datatype `x` contains, like this:

```
var x = 4;
trace (typeof x);
```

When that line of code is executed in Flash, we get the word "number" in the Output window. Notice that we provide the *typeof* operator with a value upon which to operate, but without using parentheses: `typeof x`. You might therefore wonder whether or not `x` is an *argument* of *typeof*. In fact, `x` plays the same role as an argument (it's an ancillary piece of data needed in the computation of the phrase of code), but in the context of an operator, the argument-like `x` is officially called an *operand*. An operand is an item upon which an operator operates. For example, in the expression 4 + 9, the numbers 4 and 9 are operands of the + operator.

Chapter 5, *Operators*, covers all of the ActionScript operators in detail. For now just remember that operators link phrases of code in some kind of transformation.

Putting It All Together

Let's review what you've learned. Here, again, is line one:

```
var message = "Hi there, Flash!";
```

The keyword *var* tells the interpreter that we're declaring (creating) a new variable. The word `message` is the name of our variable. The equals sign is an operator that assigns the text string ("Hi there, Flash!") to the variable `message`. The text "Hi there, Flash!" hence, becomes the value of `message`. Finally, the semicolon (`;`) tells the interpreter that we're finished with our first statement.

Line two is pretty much the same as line one:

```
var firstName = "your name here";
```

Here we're assigning the text string you typed in place of *`your name here`* to the variable `firstName`. A semicolon ends our second statement.

We then use the variables `message` and `firstName` in lines three and four:

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

The keyword *trace* signals the interpreter that we'd like some text displayed in the Output window. We pass the text we want displayed as an argument. The opening parenthesis marks the beginning of our argument. In line four, the argument itself includes two *operations*, both of which use the plus *operator*. The first operation joins its first *operand*, "Hi there, " to the value of its second operand, `firstName`. The second operation joins ", nice to meet you." to the result of the

,ch01.12472 Page 14 Thursday, April 19, 2001 12:58 PM

first operation. The closing parenthesis marks the end of our argument, and the semicolon once again indicates the end of our statement.

Blam! Your first ActionScript program. That has a nice ring to it, and it's an important landmark.

Further ActionScript Concepts

You've already been introduced to many of the fundamental elements that make up ActionScript: data, variables, operators, statements, functions, and arguments. Before we delve deeper into those topics, let's sketch out the rest of ActionScript's core features.

Flash Programs

To most computer users, *a program* is synonymous with *an application*, such as Adobe Photoshop or Macromedia Dreamweaver. Obviously, that's not what we're building when we program in Flash. Programmers, on the other hand, define a program as a collection of code (a "series of statements"), but that's only part of what we're building.

A Flash movie is more than a series of lines of code. Code in Flash is intermingled with Flash movie elements, like frames and buttons. We attach our code to those elements so that it can interact with them.

In the end, there really isn't such a thing as a Flash "program" in the classic sense of the term. Instead of complete programs written in ActionScript, we have *scripts*: code segments that give programmatic behavior to our movie, just as JavaScript scripts give programmatic behavior to HTML documents. The real product we're building is not a program but a complete movie (including its code, timelines, visuals, sound, and other assets).

Our scripts include most of what you'd see in traditional programs without the operating-system-level stuff you would write in languages like C++ or Java to place graphics on the screen or cue sounds. We're spared the need to manage the nuts 'n' bolts of graphics and sound programming, which allows us to focus most of our effort on designing the behavior of our movies.

Expressions

The statements of a script, as we've learned, contain the script's instructions. But most instructions are pretty useless without data. When we set a variable, for example, we assign some data as its value. When we use the *trace()* command, we pass data as an argument for display in the Output window. Data is the content we

,ch01.12472 Page 15 Thursday, April 19, 2001 12:58 PM

manipulate in our ActionScript code. Throughout your scripts, you'll retrieve, give, store, and generally sling around a lot of data.

In a program, any phrase of code that yields a single datum when a program runs is referred to as an *expression*. The number 7 and the string, "Welcome to my web site," are both very simple expressions. They represent simple data that will be used as-is when the program runs. As such, those expressions are called *literal expressions*, or *literals* for short.

Literals are only one kind of expression. A variable may also be an expression (variables stand in for data, so they count as expressions). Expressions get even more interesting when they are combined with operators. The expression `4 + 5`, for example, is an expression with two operands, 4 and 5, but the plus operator makes the entire expression yield the single value 9. Complex expressions may contain other, shorter expressions, provided that the entire phrase of code can still be converted into a single value.

Here we see the variable `message`:

```
var message = "Hi there, Flash!";
```

If we like, we can combine the variable expression `message` with the literal expression " How are you?" as follows:

```
message + " How are you?"
```

which becomes "Hi there, Flash! How are you?" when the program runs. You'll frequently see long expressions include shorter expressions when working with arithmetic, such as:

```
(2 + 3) * (4 / 2.5) - 1
```

It's important to be exposed to expressions early in your programming career because the term "expression" is often used in descriptions of programming concepts. For example, I might write, "To assign a value to a variable, type the name of the variable, then an equal sign followed by any expression."

Two Vital Statement Types: Conditionals and Loops

In nearly all programs, we'll use *conditionals* to add logic to our programs and *loops* to perform repetitive tasks.

Making choices using conditionals

One of the really rewarding aspects of Flash programming is making your movies smart. Here's what I mean by smart: Suppose a girl named Wendy doesn't like getting her clothes wet. Before Wendy leaves her house every morning, she looks out the window to check the weather, and if it's raining, she brings an umbrella.

,ch01.12472 Page 16 Thursday, April 19, 2001 12:58 PM

Wendy's smart. She uses basic logic—the ability to look at a series of options and make a decision about what to do based on the circumstances. We use the same basic logic when creating interactive Flash movies.

Here are a few examples of logic in a Flash movie:

- Suppose we have three sections in a movie. When a user goes to each section, we use logic to decide whether to show her the introduction to that section. If she has been to the section before, we skip the introduction. Otherwise, we show the introduction.
- Suppose we have a section of a movie that is restricted. To enter the restricted zone, the user must enter a password. If the user enters the right password, we show her the restricted content. Otherwise, we don't.
- Suppose we're moving a ball across the screen and we want it to bounce off a wall. If the ball crosses a certain point, we reverse the ball's direction. Otherwise, we let the ball continue traveling in the direction it was going.

These examples of movie logic require the use of a special type of statement called a *conditional*. Conditionals let us specify the terms under which a section of code should—or should not—be executed. Here's an example of a conditional statement:

```
if (userName == "James Bond") {
  trace ("Welcome to my web site, 007.");
}
```

The generic structure of a conditional is:

```
if (this condition is met) {
  then execute these lines of code
}
```

You'll learn more about the detailed syntax in Chapter 7, *Conditionals*. For now, remember that a conditional allows Flash to make logical decisions.

Repeating tasks using loops

Not only do we want our movies to make decisions, we want them to do tedious, repetitive tasks for us. That is, until they take over the world and enslave us and grow us in little energy pods as . . . wait . . . forget I told you that . . . ahem. Suppose you want to display a sequence of five numbers in the Output window, and you want the sequence to start at a certain number. If the starting number were 10, you could display the sequence like this:

```
trace (10);
trace (11);
trace (12);
trace (13);
trace (14);
```

,ch01.12472 Page 17 Thursday, April 19, 2001 12:58 PM

But if you want to start the sequence at 513, you'd have to retype all the numbers as follows:

```
trace (513);
trace (514);
trace (515);
trace (516);
trace (517);
```

We can avoid that retyping by making our *trace()* statements depend on a variable, like this:

```
var x = 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
```

On line 1, we set the value of the variable `x` to 1. Then at line 2, we send that value to the Output window. On line 3, we say, "Take the current value of `x`, add 1 to it, and stick the result back into our variable `x`," so `x` becomes 2. Then we send the value of `x` to the Output window again. We repeat this process three more times. By the time we're done, we've displayed a sequence of five numbers in the Output window. The beauty being that if we now want to change the starting number of our sequence, we just change the initial value of `x`. Because the rest of our code is based on `x`, the entire sequence changes when the program runs.

That's an improvement over our first approach, and it works pretty well when we're displaying only five numbers, but it becomes impractical if we want to count to 500. To perform highly repetitive tasks, we use a *loop*—a statement that causes a block of code to be repeated an arbitrary number of times. There are several types of loops, each with its own syntax. One of the most common loop types is the *while* loop. Here's what our counting example would look like as a *while* loop instead of as a series of repeated statements:

```
var x = 1;
while (x <= 5) {
  trace (x);
  x = x + 1;
}
```

The keyword *while* indicates that we want to start a loop. The expression *(x <= 5)* governs how many times the loop should execute (as long as `x` is less than or equal to 5), and the statements *trace (x);* and *x = x + 1;* are executed with each repetition (or *iteration*) of the loop. As it is, our loop saves us only 5 lines of code,

,ch01.12472 Page 18 Thursday, April 19, 2001 12:58 PM

but it could potentially save us hundreds of lines if we were counting to higher numbers. And our loop is flexible. To make our loop count to 500, we simply change the expression *(x <=5)* to *(x <=500)*:

```
var x = 1;
while (x <= 500) {
  trace (x);
  x = x + 1;
}
```

Like conditionals, loops are one of the most frequently used and important types of statements in programming.

Modular Code (Functions)

So far your longest script has consisted of four lines of code. But it won't be long before that 4 lines becomes 400 or maybe even 4,000. Sooner or later you're going to end up looking for ways to manage your code, reduce your work, and make your code easier to apply to multiple scenarios. Which is when you'll first really start to love *functions*. A function is a packaged series of statements. In practice, functions mostly serve as reusable blocks of code.

Suppose you want to write a quick script that calculates the area of a 4-sided figure. Without functions, your script might look like this:

```
var height = 10;
var width = 15;
var area = height * width;
```

Now suppose you want to calculate the area of five 4-sided figures. Your code quintuples in size:

```
var height1 = 10;
var width1 = 15;
var area1 = height1 * width1;
var height2 = 11;
var width2 = 16;
var area2 = height2 * width2;
var height3 = 12;
var width3 = 17;
var area3 = height3 * width3;
var height4 = 13;
var width4 = 18;
var area4 = height4 * width4;
var height5 = 20;
var width5 = 5;
var area5 = height5 * width5;
```

Because we're repeating the area calculation over and over, we are better off putting it in a function once and executing that function multiple times:

,ch01.12472 Page 19 Thursday, April 19, 2001 12:58 PM


```
function area(height, width){
  return height * width;
}
area1 = area(10, 15);
area2 = area(11, 16);
area3 = area(12, 17);
area4 = area(13, 18);
area5 = area(20, 5);
```

We first created the area-calculating function using the *function* statement, which defines (declares) a function just as *var* declares a variable. Then we gave our function a name, `area`, just as we give variables names. Between the parentheses, we listed the arguments that our function receives every time it's used: `height` and `width`. And between the curly braces ({ }), we included the statement(s) we want our function to execute:

```
return height * width;
```

After we create a function, we may run the code it contains from anywhere in our movie by using its name. In our example we called the *area()* function five times, passing it the `height` and `width` values it expects each time: *area(10, 15)*, *area(11, 16)*, and so on. The result of each calculation is returned to us and we store those results in the variables `area1` through `area5`. Nice and neat, and much less work than the non-function version of our code.

Don't fret if you have questions about this function example, as we'll learn more about functions in Chapter 9. For now, just remember that functions give us an extremely powerful way to create complex systems. Functions help us reuse our code and package its functionality, extending the limits of what is practical to build.

Built-in functions

Notice that functions take arguments just as the *trace()* Action does. Invoking the function *area(4, 5);* looks very much the same as issuing the *trace()* command such as *trace (x);*. The similarity is not a coincidence. As we pointed out earlier, many Actions, including the *trace()* Action, are actually functions. But they are a special type of function that is built into ActionScript (as opposed to user-defined, like our *area()* function). It is, therefore, legitimate—and technically more accurate—to say, "Call the *gotoAndStop()* function," than to say, "Execute a *gotoAndStop* Action." A built-in function is simply a reusable block of code that comes with ActionScript for our convenience. Built-in functions let us do everything from performing mathematical calculations to controlling movie clips. All the built-in functions are listed in Part III, *Language Reference*. We'll also encounter many of them as we learn ActionScript's fundamentals.

,ch01.12472 Page 20 Thursday, April 19, 2001 12:58 PM

Movie Clip Instances

With all this talk about programming fundamentals, I hope you haven't forgotten about the basics of Flash. One of the keys to visual programming in Flash is movie clip *instances*. As a Flash designer or developer, you should already be familiar with movie clips, but you may not think of movie clips as programming devices.

Every movie clip has a symbol definition that resides in the Library of a Flash movie. We can add many copies, or *instances*, of a single movie clip symbol to a Flash movie by dragging the clip from the Library onto the Stage. A great deal of advanced Flash programming is simply a matter of movie clip instance control. A bouncing ball, for example, is nothing more than a movie clip instance being repositioned on the Stage repetitively. We can alter an instance's location, size, current frame, rotation, and so forth, through ActionScript during the playback of our movie.

If you're unfamiliar with movie clips and instances, consult Flash's documentation or Help files before continuing with the rest of this book.

The Event-Based Execution Model

One final topic we should consider in our overview of ActionScript fundamentals is the *execution model*, which dictates when the code in your movie runs (is executed). You may have code attached to various frames, buttons, and movie clips throughout your movie. But when does it all actually run? To answer that question, let's take a short stroll down computing history's memory lane.

In the early days of computing, a program's instructions were executed sequentially in the order that they appeared, starting with the first line and ending with the last line. The program was meant to perform some action and then stop. That kind of program, called a *batch* program, doesn't handle the interactivity required of an *event-based* programming environment like Flash.

Event-based programs don't run in a linear fashion as batch programs do. They run continuously (in an *event loop*), waiting for things (*events*) to happen and executing code segments in response to those events. In a language designed for use with a visual interactive environment (such as ActionScript or JavaScript), the events are typically user actions such as mouseclicks or keystrokes.

When an event such as a mouseclick occurs, the interpreter sounds an alarm. A program can then react to that alarm by asking the interpreter to execute an appropriate segment of code. For example, if a user clicks a button in a movie, we could execute some code that displays a different section of the movie (classic navigation) or submits variables to a database (classic form submission).

,ch01.12472 Page 21 Thursday, April 19, 2001 12:58 PM

But programs don't react to events unless we create *event handlers*. Here's some pseudo-code that shows generally how event handlers are set up:

```
when (this event happens) {
  execute these lines of code
}
```

This is typically written in the general form:

```
on (event) {
  statements
}
```

In practice, an event handler for a button that moves the playhead to frame 200 would read:

```
on (press) {
  gotoAndStop(200);
}
```

Because event-based programs are always running an event loop, ready to react to the next event, they are like living systems. Events are a crucial part of Flash movies. Without events, our scripts wouldn't do anything—with one exception: Flash executes any code on a frame when the playhead enters that frame. The implied event is simply the playhead entering the particular frame, which is so intrinsic to Flash that no explicit event handler is required.

Events literally make things happen, which is why they come at the end of your first day of ActionScript language school. You've learned what's involved in writing scripts and what governs when those scripts will actually be executed (i.e., events). I'd say you're ready to try your first real conversation.

Building a Multiple-Choice Quiz

Now that we've explored the basic principles of ActionScript, let's apply those principles in the context of a real Flash movie. We'll start our applied study of Flash programming by creating a multiple-choice quiz using very simple programming techniques, most of which you've already learned. We'll revisit our quiz in later chapters to see how it can be improved after learning more advanced programming concepts. We'll eventually make the code more elegant so that it's easier to extend and maintain, and we'll add more features to our quiz so that it can easily handle any number of questions.

The finished *.fla* file for this quiz may be found in the online Code Depot. This is a lesson in Flash programming, not Flash production. It is assumed that you are already comfortable creating and using buttons, layers, frames, keyframes, and the

,ch01.12472 Page 22 Thursday, April 19, 2001 12:58 PM

Text tool. The quiz shows real-world applications of the following aspects of ActionScript programming:

- Variables
- Controlling the playhead of a movie with functions
- Button event handlers
- Simple conditionals
- Text field variables for on-screen display of information

Quiz Overview

Our quiz, part of which is shown in Figure 1-4, will have only two questions. Each question comes with three multiple-choice answers. Users submit their answers by clicking the button that corresponds to their desired selections. The selections are recorded in a variable so that they may be used to grade the user's score. When all the questions have been answered, the number of correct answers is tallied and the user's score is displayed.

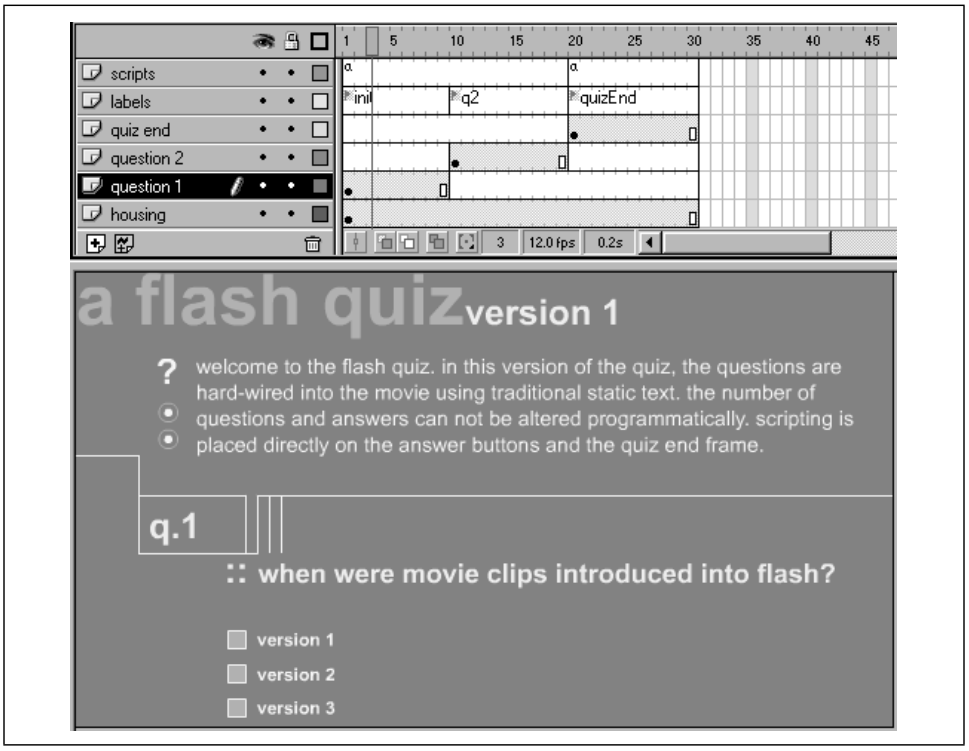

Figure 1-4. A Flash quiz

,ch01.12472 Page 23 Thursday, April 19, 2001 12:58 PM

Building the Layer Structure

When building Flash movies, it's important to organize your content into manageable divisions by keeping different content elements on individual layers. Layering content is a good production technique in general, but it is essential in Flash programming. In our quiz, and in the vast majority of our scripted movies, we'll keep all our timeline scripts on a single isolated layer, called *scripts*. I keep the *scripts* layer as the first one in my layer stack so that it's easy to find.

We'll also keep all our frame labels on a separate layer, called (surprise, surprise) *labels*. The *labels* layer should live beneath the *scripts* layer on all your timelines. In addition to these two standard layers (*scripts* and *labels*), our quiz movie has a series of content layers on which we'll isolate our various content assets.

Start building your quiz by creating and naming the following layers and arranging them in the order that they appear here:

scripts
labels
quiz end
question 2
question 1
housing

Now add 30 frames to each of your layers. Your timeline should look like the one in Figure 1-5.

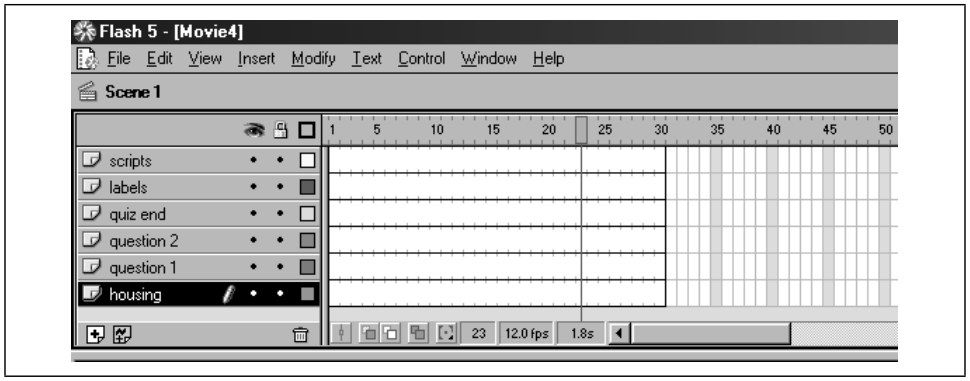

Figure 1-5. Quiz timeline initial setup

Creating the Interface and Questions

Before we get into the scripts that run the quiz, we need to set up the questions and the interface that will let the user proceed through the quiz.

,ch01.12472 Page 24 Thursday, April 19, 2001 12:58 PM

Here are the steps you should follow:

1. With frame 1 of the *housing* layer selected, use the Text tool to type your quiz title directly on the Stage.
2. At frame 1 of the *question 1* layer, add the question number "1" and the text for Question 1, "When were movie clips introduced into Flash?" Leave room for the answer text and buttons below your question.
3. Create a simple button that looks like a checkbox or radio button and measures no higher than a line of text (see Figure 1-6).
4. Below your question text (still on the *question 1* layer), add the text of your three multiple-choice answers: "Version 1," "Version 2," and "Version 3," each on its own line.
5. Next to each of your three answers, place an instance of your checkbox button.
6. We'll use Question 1 as a template for Question 2. Select the first frame of the *question 1* layer and choose Edit → Copy Frames.
7. Select frame 10 of the *question 2* layer and choose Edit → Paste Frames. A duplicate of your first question appears on the *question 2* layer at frame 10.
8. While still in frame 10 of the *question 2* layer, change the question number from "1" to "2" and change the text of the question to, "When was MP3 audio support added to Flash?" Change the multiple-choice answers to "Version 3," "Version 4," and "Version 5."
9. Finally, to prevent Question 1 from appearing underneath Question 2, add a blank keyframe at frame 10 of the *question 1* layer.

Figure 1-6 shows the Flash movie after you've added the first question to the quiz. Figure 1-7 shows how your timeline will look after you've added the two questions to the quiz.

Initializing the Quiz

Our first order of business in our quiz script (and in most scripts) is to create the main timeline variables we'll use throughout our movie. In our quiz we do this on the first frame of the movie, but in other movies we'll normally do it after preloading part or all of the movie. Either way, we want to initialize our variables before any other scripting occurs. Once our variables are defined, we invoke the *stop()* function to keep the user paused on the first frame (where the quiz starts).

For more complex movies, we may also set the initial conditions by calling functions and assigning variable values in preparation for the rest of the movie. This step is known as *initialization*. Functions that start processes in motion or define the initial conditions under which a system operates are frequently named *init*.

,ch01.12472 Page 25 Thursday, April 19, 2001 12:58 PM

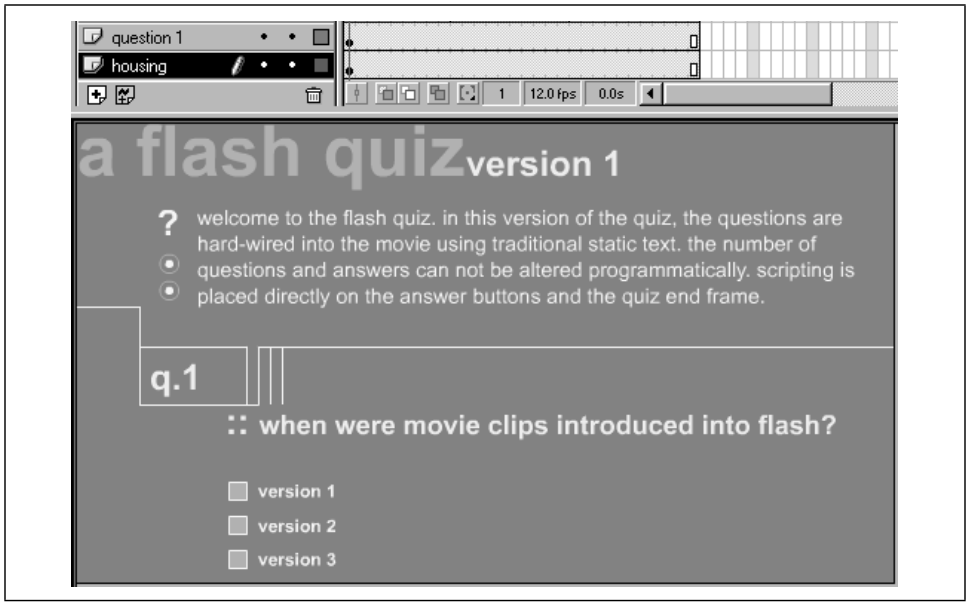

Figure 1-6. Quiz title and Question 1

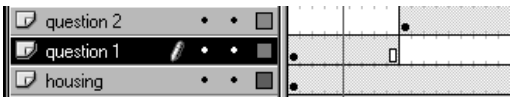

Figure 1-7. Quiz timeline with two questions

Our quiz *init* code, shown in Example 1-1, is attached to frame 1 of the *scripts* layer of our movie.

Example 1-1. Init Code for Quiz

```
// Init main timeline variables
var q1answer;          // User's answer for question 1
var q2answer;          // User's answer for question 2
var totalCorrect = 0;  // Counts number of correct answers
var displayTotal;      // Text field for displaying user's score

// Stop the movie at the first question
stop();
```

Line 1 of our *init* sequence is a *code comment*. Code comments are notes that you add in your code to explain what's going on. A single-line comment starts with two forward slashes and a space, which is then followed by a line of text:

```
// This is a comment
```

Notice that comments can be placed on the same line as your code, like this:

```
x = 5; // This is also a comment
```

,ch01.12472 Page 26 Thursday, April 19, 2001 12:58 PM

Line 2 of Example 1-1 creates a variable named `q1answer`. Recall that to create a variable we use the *var* keyword followed by a variable name, as in:

```
var favoriteColor;
```

So, the second through fifth lines of our code declare the variables we'll need, complete with comments explaining their purpose:

- `q1answer` and `q2answer` will contain the value of the user's answer (1, 2, or 3, indicating which of the three multiple-choice options was selected for each question). We'll use these values to check whether the user answered the questions correctly.
- `totalCorrect` will be used at the end of the quiz to tally the number of questions that the user answered correctly.
- `displayTotal` is the name of the text field that we'll use to show the value of `totalCorrect` on screen.

Take a closer look at Line 4 of Example 1-1:

```
var totalCorrect = 0;  // Counts number of correct answers
```

Line 4 performs double duty; it first declares the variable `totalCorrect` and then assigns the value 0 to that variable using the assignment operator, `=`. We want `totalCorrect` to default to 0 in case the user hasn't answered any of the questions correctly. The other variables don't need default values because they are all set explicitly during the quiz.

After our variables have been defined, we call the *stop()* function, which halts the playback of the movie on frame 1, where the quiz begins:

```
// Stop the movie at the first question
stop();
```

The *stop()* function has the exact same effect as any *stop* Action you may have used in Flash 4 or earlier (it pauses the playhead in the current frame).

Observe, again, the use of the comment before the *stop()* function call. That comment explains the intended effect of the code that follows. Comments are optional, but they help clarify our code if we leave it for a while and need a refresher when we return or if we pass our code to another developer. Comments also make code easy to scan, which is important during debugging.

Now that you know what our *init* code does, let's add it to our quiz movie:

1. Select Frame 1 of the *scripts* layer.
2. Choose Window → Actions. The Frame Actions panel appears.

,ch01.12472 Page 27 Thursday, April 19, 2001 12:58 PM

3. Make sure you're using Expert Mode, which can be set as a permanent preference under Edit → Preferences.
4. Into the right side of the Frame Actions panel, type the *init* code as shown earlier in Example 1-1.

Variable Naming Styles

By now you've seen quite a few variable names, and you may be wondering about the capitalization. If you've never programmed before, a capital letter in the middle of a word, as in `firstName`, or `totalCorrect`, may seem odd. Capitalizing the second word (and any following words) of a variable name visually demarcates the words within that name. We use this technique because spaces and dashes aren't allowed in a variable name. But don't capitalize the first letter of a variable name—an initial capital letter is conventionally used to name object classes, not variables.

If you use underscores instead of capital letters to separate words in variables, as in `first_name` and `total_correct`, be consistent. Don't use `firstName` for some variables and `second_name` for others. Use one of these styles so that other programmers will find your code understandable. Variable names in some languages are case-sensitive, meaning that `firstName` and `firstname` would be considered two different variables. ActionScript, however, treats them as the same thing. But it's bad form to use two different cases to refer to the same variable; if you call a variable `xPOS`, don't refer to it elsewhere as `xpos`.

Always give your variables and functions meaningful names that help you remember what they are for. Avoid useless names like "foo," and use single-letter variables, such as "x" or "i" only for simple things like the index (i.e., counting variable) in a loop.

Adding Frame Labels

We've got our quiz's *init* script done and our questions built. We should now add some frame labels so that we can control the playback of our quiz.

In order to step the user through our quiz one question at a time, we've separated the content for Question 1 and Question 2 into frames 1 and 10. By moving the playhead to those keyframes, we'll create a slide show effect, where each slide contains a question. We know that Question 2 is on frame 10, so when we want to display Question 2, we can call the *gotoAndStop()* function like this:

```
gotoAndStop(10);
```

which would cause the playhead to advance to frame 10, the location of Question 2. A sensible piece of code, right? Wrong! Whereas using the specific number 10

,ch01.12472 Page 28 Thursday, April 19, 2001 12:58 PM

with our *gotoAndStop()* function works, it isn't flexible. If, for example, we added five frames to the timeline before frame 10, Question 2 would suddenly reside at frame 15, and our *gotoAndStop(10)* command would not bring the user to the correct frame. To allow our code to work even if the frames in our timeline shift, we use *frame labels* instead of frame numbers. Frame labels are expressive names, such as `q2` or `quizEnd`, by which we can refer to specific points on the timeline. Once a point is labeled, we can use the label to refer to the frame by name instead of by number.

The flexibility of frame labels is indispensable. I hardly ever use frame numbers with playback-control functions like *gotoAndStop()*. Let's add all the labels we'll need for our quiz now, so that we can use them later to walk the user through the quiz questions:

1. On the *labels* layer, click frame 1.
2. Select Modify → Frame. The Frame panel appears.
3. In the Label text field, type **`init`**.
4. At frame 10 of the *labels* layer, add a blank keyframe.
5. In the Frame panel, in the Label text field, type **`q2`**.
6. At frame 20 of the *labels* layer, add a blank keyframe.
7. In the Frame panel, in the Label text field, type **`quizEnd`**.

Scripting the Answer Buttons

Our questions are in place, our variables have been initialized, and our frames have been labeled. If we were to test our movie now, we'd see Question 1 appear with three answer buttons that do nothing when clicked and no way for the user to get to Question 2. We need to add some code to the answer buttons so that they will advance the user through the quiz and keep track of his answers along the way.

For convenience, we'll refer to the multiple-choice buttons as button 1, button 2, and button 3, as shown in Figure 1-8.

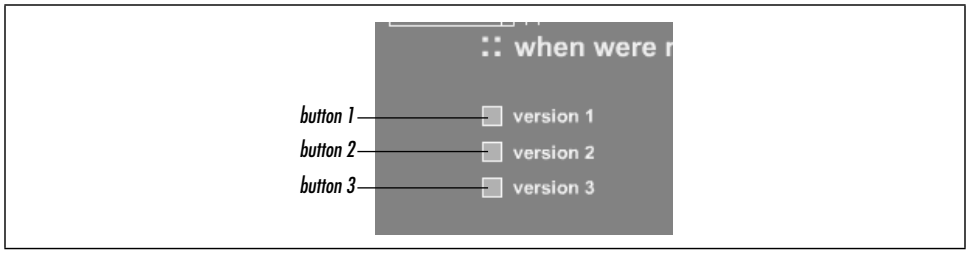

Figure 1-8. The answer buttons

,ch01.12472 Page 29 Thursday, April 19, 2001 12:58 PM

Our three buttons get very similar scripts. Example 1-2 through Example 1-4 show the code for each button.

Example 1-2. Code for Question 1, Button 1

```
on (release) {
  q1answer = 1;
  gotoAndStop ("q2");
}
```

Example 1-3. Code for Question 1, Button 2

```
on (release) {
  q1answer = 2;
  gotoAndStop ("q2");
}
```

Example 1-4. Code for Question 1, Button 3

```
on (release) {
  q1answer = 3;
  gotoAndStop ("q2");
}
```

The button code consists of two statements (lines 2 and 3) that are executed only when a mouseclick is detected. In natural language, the code for each button says, "When the user clicks this button, make a note that he chose answer 1, 2, or 3, then proceed to Question 2." Here's how it works.

Line 1 is the beginning of an *event handler*:

```
on (release) {
```

The event handler waits patiently for the user to click button 1. Recall that an event handler listens for things (such as mouseclicks) that happen while the movie is running. When an event occurs, the code contained in the appropriate handler is executed.

Let's dissect the event handler that begins on line 1. The keyword *on* signals the start of the event handler. (If the word *on* seems a little awkward to you, think of it as *when* until you're comfortable with it.) The keyword *release*, enclosed in parentheses, indicates the *type of event* that the event handler is listening for; in this case, we're listening for a *release* event, which occurs when the user clicks and releases the mouse over the button. The opening curly brace ({) marks the beginning of the block of statements that should be executed when the *release* event occurs. The end of the code block is marked by a closing curly brace (}) on line 4, which is the end of the event handler.

Line 2 is the first of the statements that will be executed when the *release* event occurs. The code in line 2 should be getting quite familiar to you:

```
q1answer = 1;
```

,ch01.12472 Page 30 Thursday, April 19, 2001 12:58 PM

It sets the variable `q1answer` to 1 (the other answer buttons set it to 2 or 3). The `q1answer` variable stores the user's answer for the first question. Once we have recorded the user's answer for Question 1, we advance to Question 2 via line 3 of our button code:

```
gotoAndStop ("q2");
```

Line 3 calls the *gotoAndStop()* function, passing it the frame label "q2" as an argument, which advances the playhead to the frame `q2` where Question 2 appears.

Now that you know how the button code works, let's add it to the Question 1 buttons:

1. With the Actions panel open, select button 1 on the Stage. The Frame Actions title changes to Object Actions. Any code you add now will be attached to button 1 (the selected object on the Stage).
2. Into the right side of the Actions panel, type the code from Example 1-2.
3. Repeat steps 1 and 2 to add button code to buttons 2 and 3. On button 2, set `q1answer` to 2; on button 3, set `q1answer` to 3, as shown in Example 1-3 and Example 1-4.

The code for the Question 2 buttons is structurally identical to that of the Question 1 buttons (we change only the name of the answer variable and the destination of the *gotoAndStop()* call). Example 1-5 shows the code for button 1 of Question 2.

Example 1-5. Code for Question 2, Button 1

```
on (release) {
  q2answer = 1;
  gotoAndStop ("quizEnd");
}
```

We use the variable `q2answer` instead of `q1answer` because we want the buttons to keep track of the user's selection for Question 2. We use "quizEnd" as the argument for our *gotoAndStop()* function call to advance the playhead to the end of the quiz (i.e., the frame labeled `quizEnd`) after the user answers Question 2.

Let's add the button code for the Question 2 buttons:

1. Click on frame 10 of the *question 2* layer.
2. Click on button 1.
3. Into the Actions panel, type the code from Example 1-5.
4. Repeat steps 2 and 3 to add button code to buttons 2 and 3. On button 2, set `q2answer` to 2. On button 3, set `q2answer` to 3.

Having just added button code to six buttons, you will no doubt have noticed how repetitive the code is. The code on each button differs from the code on the others

,ch01.12472 Page 31 Thursday, April 19, 2001 12:58 PM

by only a few text characters. That's not exactly efficient programming. Our button code cries out for some kind of centralized entity that records the answer and advances to the next screen in the quiz. In Chapter 9 we'll see how to centralize our code with *functions*.

Building the Quiz End

Our quiz is nearly complete. We now have two questions working with an answer-tracking script that lets the user answer the questions and progress through the quiz. We still need a quiz-ending screen where we tell the user how well he fared.

To build our quiz-end screen, we need to do some basic Flash production and some scripting. Let's do the production first:

1. At frame 20 of the *question 2* layer, add a blank keyframe. This prevents Question 2 from appearing underneath the contents of our quiz-end screen.
2. At frame 20 of the *quiz end* layer, add a blank keyframe.
3. While you're still on that frame, put the following text on the Stage: "Thank you for taking the quiz. Your final score is: /2." Make sure to leave a decent amount of blank space between "is:" and "/2." We'll put the user's score there.
4. At frame 20 of the *scripts* layer, add a blank keyframe.

That takes care of the production work for our quiz-end screen. Your end screen should look something like the one shown in Figure 1-9.

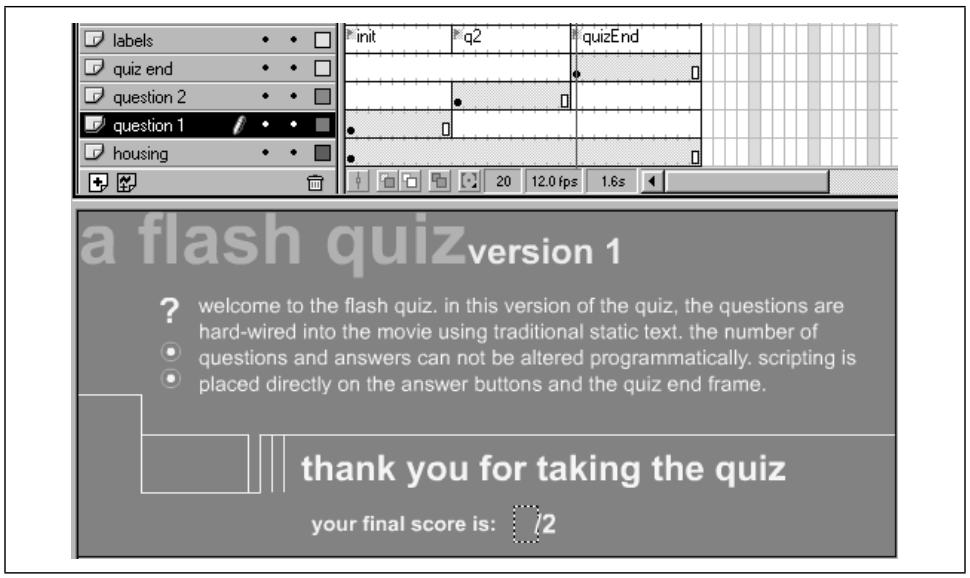

Figure 1-9. Judgment day

,ch01.12472 Page 32 Thursday, April 19, 2001 12:58 PM

Now let's work on the quiz-end script. When the playhead lands on our `quizEnd` frame, we want to calculate the user's score. We need a calculation script to execute when the playhead reaches frame 20. Because any script placed on a keyframe in the timeline is automatically executed when the playhead enters that frame, we can simply attach our calculation script to the keyframe we added at frame 20 of the *scripts* layer.

In the calculation script, we first determine the user's score, and then we display that score on the screen:

```
// Tally up the user's correct answers
if (q1answer == 3){
  totalCorrect = totalCorrect + 1;
}
if (q2answer == 2){
  totalCorrect++;
}
// Show the user's score in an on-screen text field
displayTotal = totalCorrect;
```

Lines 1 and 8 are code comments that summarize the functionality of the two sections of the script. On line 2, the first of two conditionals in our calculation script begins. In it, we see our `q1answer` variable put to use:

```
if (q1answer == 3){
```

The keyword *if* tells the interpreter we're about to provide a list of statements that should be executed only if a certain condition is met. The terms of that condition are described in the parentheses that follow the *if* keyword: *(q1answer == 3)*, and the opening curly brace begins the list of statements to be conditionally executed. Therefore, line 2 translates into, "If the value of `q1answer` is equal to 3, then execute the statements contained in the following curly braces."

But how exactly does the condition *q1answer == 3* work? Well, let's break the phrase down. We recognize `q1answer` as the variable in which we've stored the user's answer to Question 1. The number 3 indicates the correct answer to Question 1, because movie clips first appeared in Flash 3. The double equal sign (==) between our variable and the number 3 is the *equality* comparison operator, which compares two expressions. If the expression on its left (`q1answer`) equals the one on its right (3), our condition is met, and the statements within the curly braces are executed. If not, our condition is not met, and the statements within the curly braces are skipped.

Flash has no way of knowing the right answers to our quiz questions. Checking if `q1answer` is equal to 3 is our way of telling Flash to check if the user got Question 1 right. If he did, we tell Flash to add one to his total score as follows:

```
totalCorrect = totalCorrect + 1;
```

,ch01.12472 Page 33 Thursday, April 19, 2001 12:58 PM

Line 3 says, "Make the new value of `totalCorrect` equal to the old value of `totalCorrect` plus one," (i.e., *increment* `totalCorrect`). Incrementing a variable is so common that it has its own special operator, ++.

So instead of using this code:

```
totalCorrect = totalCorrect + 1;
```

We normally write:

```
totalCorrect++;
```

which does exactly the same thing, but more succinctly.

At line 4, we end the block of statements to execute if our first condition is met:

```
}
```

Lines 5 through 7 are another condition:

```
if (q2answer == 2){
  totalCorrect++;
}
```

Here we're checking whether the user answered Question 2 correctly (MP3 audio support first appeared in Flash 4). If the user chose the second answer, we add one to `totalCorrect` using the increment operator ++.

Because there are only two questions in our quiz, we're done tallying the user's score. For each question that the user answered correctly, we added one to `totalCorrect`, so `totalCorrect` now contains the user's final score. The only thing left is to show the user his score, via line 9, the last line of our quiz-end script:

```
displayTotal = totalCorrect;
```

You already know enough about variables to guess that the statement on line 9 assigns the value of `totalCorrect` to the variable `displayTotal`. But how does that make the score appear on screen? So far, it doesn't. In order to make the score appear on screen, we need to create a special kind of variable called a *text field* variable that has a physical representation on the screen. Let's make one now so you can see how it works:

1. Select the Text tool.
2. On the *quiz end* layer, click frame 20.
3. Place your pointer just before the text "/2" that you created earlier, then click the Stage.
4. Drag out a text box big enough to hold a single number.
5. Choose Text → Options.

,ch01.12472 Page 34 Thursday, April 19, 2001 12:58 PM

6. In the Text Options panel, change Static Text to Dynamic Text.
7. In the Variable text field, type **`displayTotal`**.

The variable `displayTotal` now has a screen representation. If we change `displayTotal` in our script, the corresponding text field variable will be updated on the screen.

Testing Our Quiz

Well, that's it. Our quiz is finished. You can now check whether the quiz works using Control → Test Movie. Click on the answers in different combinations to see if your quiz is keeping score correctly. You can even create a restart button by attaching the following code to a new button:

```
on (release) {
  gotoAndStop("init")
}
```

Because `totalCorrect` is set to 0 in the code on the `init` frame, the score will reset itself each time you send the playhead to `init`.

If you find that your quiz isn't working, try comparing it with the sample quiz provided at the online Code Depot. You may also want to investigate the troubleshooting techniques described in Chapter 19, *Debugging*.

Onward!

So how does it feel? You've learned a bunch of phrases, some grammar, some vocabulary, and even had a drawn-out conversation with Flash (the multiple-choice quiz). Quite a rich first day of language school, I'd say.

As you can see, there's a lot to learn about ActionScript, but you can also do quite a bit with just a little knowledge. Even the amount you know now will give you plenty to play around with. Over the rest of this book, we'll reinforce the fundamentals you've learned by exploring them in more depth and showing them in concert with real examples. Of course, we'll also cover some topics that haven't even been introduced yet.

Remember: think communication, think cooperation, and speak clearly. And if you find yourself doing any fantastically engaging work or art that you'd like to share with others, send it over to me at *http://ww.moock.org/webdesign/flash/contact.html*.

Now that you have a practical frame of reference, you'll be better able to appreciate and retain the foundational knowledge detailed over the next few chapters. It will give you a deeper understanding of ActionScript, enabling you to create more complex movies.