2

Variables

In a typical scripted movie, we have to track and manipulate everything from frame numbers to a user's password to the velocity of a photon torpedo fired from a spaceship. In order to manage and retrieve all that information, we need to store it in *variables*, the primary information-storage containers of ActionScript.

A variable is a like a bank account that, instead of holding money, holds information (*data*). Creating a new variable is like setting up a new account; we establish a place to store something we'll need in the future. And just as every bank account has an account number, every variable has a name associated with it that is used to access the data in the variable.

Once a variable is created, we can put new data into it as often as we want—much like depositing money into an account. Or we can find out what's in a variable using the variable's name—much like checking an account balance. If we no longer need our variable, we can "close the account" by deleting the variable.

The key feature to note is that variables let us refer to data that either changes or is calculated when a movie plays. Just as a bank account's number remains the same even though the account balance varies, a variable's name remains fixed even though the data it contains may change. Using that fixed reference to access changing content, we can perform complex calculations, keep track of cards in a card game, save guest book entries, or send the playhead to different locations based on changing conditions.

Is that a gleam of excitement I see in your eye? Good, I thought I might have lost you with all that talk about banks. Let's start our exploration of variables by seeing how to create them.

Creating Variables (Declaration)

Creating a variable is called *declaration*. Declaration is the "open an account" step of our bank metaphor, where we formally bring the variable into existence. When a variable is first declared, it is empty—a blank page waiting to be written upon. In this state, a variable contains a special value called `undefined` (indicating the absence of data).

To declare a new variable, we use the *var* statement. For example:

```
var speed;
var bookTitle;
var x;
```

The word *var* tells the interpreter that we're declaring a variable, and the text that follows, such as `speed`, `bookTitle`, or `x`, becomes our new variable's name. We can create variables anywhere we can attach code: on a keyframe, a button, or a movie clip.

We can also declare several variables with one *var* statement, like this:

```
var x, y, z;
```

However, doing so impairs our ability to add comments next to each variable.

Once a variable has been created, we may assign it a value, but before we learn how to do that, let's consider some of the subtler details of variable declaration.

Automatic Variable Creation

Many programming languages require variables to be declared before data may be deposited into them; failure to do so would cause an error. ActionScript is not that strict. If we assign a value to a variable that does not exist, the interpreter will create a new variable for us. The bank, to continue that analogy, automatically opens an account when you try to make your first deposit.

This convenience comes at a cost, though. If we don't declare our variables ourselves, we have no central inventory to consult when examining our code. Furthermore, explicitly declaring a variable with a *var* statement can sometimes yield different results than allowing a variable to be declared *implicitly* (i.e., automatically). It's safest to declare first and use later (i.e., *explicit declaration*), as shown throughout this book.

Legal Variable Names

Before running off to make any variables, be aware that variable names:

- Must be composed exclusively of letters, numbers, and underscores. (No spaces, hyphens, or punctuation allowed.)
- Must start with a letter or an underscore.
- Must not exceed 255 characters. (Okay, okay, that's a lie, but reevaluate your naming scheme if your variable names exceed 255 characters.)
- Are case-insensitive (upper- and lowercase are treated identically but you should be consistent nonetheless).

These are legal variable names:

```
var first_name;
var counter;
var reallyLongVariableName;
```

These are illegal variable names that would cause errors:

```
var 1first_name;                // Starts with a number
var variable name with spaces;  // Contains spaces
var another-illegal-name;       // Contains a hyphen
```

Creating dynamically named variables

Although you'll rarely, if ever, use dynamically created variable names, it's possible to generate the name of a variable programmatically. To create a variable name from any expression, use the *set* statement. For example, here we assign the value "bruce" to `player1name`:

```
var i = 1;
set ("player" + i + "name", "bruce");
```

Arrays and objects, discussed in later chapters, provide us with a much more powerful means of tracking dynamically named data and should be used instead of dynamic variable names.

Declare Variables at the Outset

It's good practice to declare your variables at the beginning of every movie's main script space, which is usually the first keyframe that comes after a movie's preloader. Be sure to add a comment explaining each variable's purpose for easy identification later. The beginning of a well-organized script might look like this:

```
// ^^^^^^^^^^^^^^^^^^^^
// Initialize variables
// ^^^^^^^^^^^^^^^^^^^^
var ballSpeed;   // Velocity of ball, max 10
var score;       // Player's current score
var hiScore;     // High score (not saved between sessions)
var player1;     // Name of player 1, supplied by user
```

We can give variables an initial value at the same time we create them, as follows:

```
var ballSpeed = 5;   // Velocity of ball, max 10
var score = 0;       // Player's current score
var hiScore = 0;     // High score (not saved between sessions)
```

Assigning Variables

Now comes the fun part—putting some data into our variables. If you're still playing along with the bank analogy, this is the "deposit money into our account" step. To assign a variable a value, we use:

```
variableName = value;
```

where *variableName* is the name of a variable, and *value* is the data we're assigning to that variable. Here's an applied example:

```
bookTitle = "ActionScript: The Definitive Guide";
```

On the left side of the equal sign, the word `bookTitle` is the variable's *name* (its *identifier*). On the right side of the equal sign, the phrase "ActionScript: The Definitive Guide" is the variable's *value*—the datum you're depositing. The equal sign itself is called the *assignment* operator. It tells Flash that you want to assign (i.e., deposit) whatever is on the right of the equal sign to the variable shown on the left. If the variable on the left doesn't exist yet, Flash creates it (though relying on the interpreter to implicitly create variables isn't recommended).

Here are two more variable assignment examples:

```
speed = 25;
output = "thank you";
```

The first example assigns the integer 25 to the variable `speed`, showing that variables can contain numbers as well as text. We'll see shortly that they can contain other kinds of data as well. The second example assigns the text "thank you" to the variable `output`. Notice that we use straight double quotation marks (`" "`) to delimit a text string in ActionScript.

Now let's look at a slightly more complicated example that assigns `y` the value of the expression 1 + 5:

```
y = 1 + 5;
```

When the statement `y = 1 + 5;` is executed, 1 is first added to 5, yielding 6, and then 6 is assigned to `y`. The expression on the right side of the equal sign is *evaluated* (calculated or resolved) before setting the variable on the left side equal to that result. Here we assign an expression that contains the variable `y` to another variable, `z`:

```
z = y + 4;
```

Once again, the expression on the right of the equal sign is evaluated and the result is then assigned to `z`. The interpreter retrieves the current value of `y` (it checks its account balance, so to speak) and adds 4 to it. Because the value of `y` is 6, `z` will be set to 10.

The syntax to assign any data—whether numbers, text, or any other type—to a variable is similar regardless of the datatype. For example, we haven't studied arrays yet, but you should already recognize the following as a variable assignment statement:

```
myList = ["John", "Joyce", "Sharon", "Rick", "Megan"];
```

As before, we put the variable name on the left, the assignment operator (the equal sign) in the middle, and our desired value on the right.

To assign the same value to multiple variables in a hurry, we may piggyback assignments alongside one another, like this:

```
x = y = z = 10;
```

Variable assignment always works from right to left. The preceding statement assigns 10 to `z`, then assigns the value of `z` to `y`, then assigns the value of `y` to `x`.

Changing and Retrieving Variable Values

After we've created a variable, we may assign and reassign its value as often as we like, as shown in Example 2-1.

Example 2-1. Changing Variable Values

```
var firstName;                  // Declare the variable firstName
firstName = "Graham";           // Set the value of firstName
firstName = "Gillian";          // Change the value of firstName
firstName = "Jessica";          // Change firstName again
firstName = "James";            // Change firstName again
var x = 10;                     // Declare x and assign a numeric value
x = "loading...please wait..."; // Assign x a text value
```

Notice that we changed the variable `x`'s *datatype* from numeric to text data by simply assigning it a value of the desired type. Some programming languages don't allow the datatype of a variable to change but ActionScript does.

Of course, creating variables and assigning values to them is useless if you can't retrieve the values later. To retrieve a variable's value, simply use the variable's name wherever you want its value to be used. Anytime a variable's name appears (except in a declaration or on the left side of an assignment statement), the name is converted to the variable's value. Here are some examples:

```
newX = oldX + 5;  // Set newX to the value of oldX plus 5
ball._x = newX;   // Set the horizontal position of the
                  // ball movie clip to the value of newX
trace(firstName); // Display the value of firstName in the Output window
```

Note that in the expression `ball._x`, `ball` is a movie clip's name, and the `._x` indicates its x-coordinate property (i.e., horizontal position on stage). We'll learn more about properties later. The last line, `trace(firstName)`, displays a variable's value while a script is running, which is handy for debugging your code.

Checking Whether a Variable Has a Value

Occasionally we may wish to verify that a variable has been assigned a value before we make reference to it. As we learned earlier, a variable that has been declared but never assigned a value contains the special "non-value," `undefined`. To determine whether a variable has been assigned a value, we compare that variable's value to the `undefined` keyword. For example:

```
if (someVariable != undefined) {
  // Any code placed here is executed only if someVariable has a value
}
```

Note the use of the *inequality operator*, `!=`, which determines whether two values are *not* equal.

Types of Values

The data we use in ActionScript programming comes in a variety of types. So far we've seen numbers and text, but other types include Booleans, arrays, functions, and objects. Before we cover each datatype in detail, let's examine some datatype issues that specifically relate to variable usage.

Automatic Typing

Any ActionScript variable can contain any type of data, which may seem unremarkable, but the ability to store *any* kind of data in *any* variable is actually a bit unusual. Languages like C++ and Java use *typed* variables; each variable can accept only one type of data, which must be specified when the variable is declared. ActionScript variables are *automatically* typed—when we assign data to a variable, the interpreter sets the variable's datatype for us.

Not only can ActionScript variables contain any datatype, they can also dynamically *change* datatypes. If we assign a variable a new value that has a different type than the variable's previous value, the variable is automatically retyped. So the following code is legal in ActionScript:

```
x = 1;                  // x is a number
x = "Michael";          // x is now a string
x = [4, 6, "hello"];    // x is now an array
x = 2;                  // x is a number again
```

In languages like C++ or Java that do not support automatic retyping, data of the wrong type would be converted to the variable's existing datatype (or would cause an error if conversion could not be performed). Automatic and dynamic typing have some important ramifications that we'll consider in the following sections.

Automatic Value Conversion

In some contexts, ActionScript expects a specific type of data. If we use a variable whose value does not match the expected type, the interpreter attempts to convert the data. For example, if we use a text variable where a number is needed, the interpreter will try to convert the variable's text value to a numeric value for the sake of the current operation. In Example 2-2, `z` is set to 2. Why? Because the subtraction operator expects a number, so the value of `y` is converted from the string "4" to the number 4, which is subtracted from 6 (the value of `x`), yielding the result 2.

Example 2-2. Automatic String-to-Number Conversion

```
x = 6;      // x is a number, 6
y = "4";    // y is a string, "4"
z = x - y;  // This sets z to the number 2
```

Conversely, if we use a numeric variable where a string is expected, the interpreter attempts to convert the number to a string. In Example 2-3, `z` is set to the string "64", not the number 10. Why? Because the second operand in the expression `x + y` is a string. Therefore, the (+) performs string concatenation instead of mathematical addition. The value of `x` (6) is converted to the string "6" and then concatenated with the string "4" (the value of `y`), yielding the result "64".

Example 2-3. Automatic Number-to-String Conversion

```
x = 6;      // x is a number, 6
y = "4";    // y is a string, "4"
z = x + y;  // This sets z to the string "64"
```

The automatic type conversion that occurs when evaluating a variable as part of an expression is performed on a *copy* of the variable's data—it does not affect the original variable's type. A variable's type changes only when the variable is assigned a data value that does not match its previous value's type. So at the conclusion of Example 2-2 and Example 2-3, `y` remains a string, and `x` remains a number.

Notice that the operator on line 3 (– in Example 2-2, + in Example 2-3), has a profound impact on the value assigned to `z`. In Example 2-2 the string "4" becomes the number 4, whereas in Example 2-3 the opposite occurs (the number 6 becomes the string "6"), because the rules for datatype conversion are different for the + operator than for the – operator. We'll cover data conversion rules in Chapter 3, *Data and Datatypes*, and operators in Chapter 5, *Operators*.

Determining the Type Manually

Automatic datatyping and conversion can be convenient, but as Example 2-2 and Example 2-3 illustrate, may also produce unexpected results. Before performing commands that operate on mixed datatypes, you may wish to determine a variable's datatype using the *typeof* operator:

```
productName = "Macromedia Flash";  // String value
trace(typeof productName);         // Displays: "string"
```

Once we know a variable's type, we can proceed conditionally. Here, for example, we check whether a variable is a number before proceeding:

```
if (typeof age == "number"){
  // okay to carry on
} else {
  trace ("Age isn't a number");  // Display an error message
}
```

For full details on the *typeof* operator, see Chapter 5.

Variable Scope

Earlier we learned how to create variables and retrieve their values using variables attached to a single frame of the main timeline of a Flash document. When a document contains multiple frames and multiple movie clip timelines, variable creation and value retrieval becomes a little more complicated.

To illustrate why, let's consider several scenarios.

Scenario 1

Suppose we were to create a variable, `x`, in frame 1 of the main timeline. After creating `x`, we set its value to 10:

```
var x;
x = 10;
```

Then, in the next frame (frame 2), we attach the following code:

```
trace(x);
```

When we play our movie, does anything appear in the Output window? We created our variable in frame 1, but we're attempting to retrieve its value in frame 2; does our variable still exist? Yes.

When you define a variable on a timeline, that variable is accessible from all the other frames of that timeline.

Scenario 2

Suppose we create and set `x` as we did in Scenario 1, but instead of placing the variable-setting code on frame 1 directly, we place it on a button in frame 1. Then, on frame 2, we attach the same code as before:

```
trace(x);
```

Does Scenario 2 also work? Yes. Because `x` is attached to our button, and our button is attached to the main timeline, our variable is indirectly attached to the main timeline. We may, therefore, access the variable from frame 2 as we did before.

Scenario 3

Suppose we create a variable named `secretPassword` on frame 1 of the main timeline. When the movie plays, the user must guess the password in order to gain access to a special section of the movie.

In addition to declaring `secretPassword` on frame 1, we create a function that compares the user's guess to the real password. Here's our code:

```
var secretPassword;
secretPassword = "yppah";

function checkPassword() {
  if (userPassword == secretPassword) {
    gotoAndStop("accessGranted");
  } else {
    gotoAndStop("accessDenied");
  }
}
```

Suppose we ask the user to enter her password on frame 30. She enters a password into an input text field variable named `userPassword`, which we compare to our `secretPassword` variable using the *checkPassword()* function on frame 1. If our password-checking code is defined on frame 1, but `userPassword` isn't defined until frame 30, does the `userPassword` variable exist when we call our *checkPassword()* function?

The answer, again, is yes. Even though `userPassword` is defined on a later frame than our *checkPassword()* function, it is still part of the same timeline.

Any variable declared on a timeline is available to all the scripts of its timeline for as long as that timeline exists.

Variable Accessibility (Scope)

The three scenarios presented earlier explore issues of *scope*. A variable's scope describes when and where the variable can be manipulated by the code in a movie. A variable's scope defines its life span and its accessibility to other blocks of code in our scripts. To determine a variable's scope, we must answer two questions: (a) how long does the variable exist? and (b) from where in our code can we set or retrieve the variable's value?

In traditional programming, variables are often broken into two general scope categories: *global* and *local*. Variables that are accessible throughout an entire program are called *global variables*. Variables that are accessible only to limited sections of a program are called *local variables*. Though Flash supports conventional local variables, it does not support true global variables. Let's find out why.

Movie Clip Variables

As we saw in the three earlier scenarios, a variable defined on a timeline is available to all the scripts on that timeline—from the first frame to the last frame—whether the variable is declared on a frame or a button. But what happens if we have more than one timeline in a movie, as described in Scenario 4?

Scenario 4

Suppose we have two basic geometric shapes, a square and a circle, defined as movie clip symbols.

On frame 1 of the square clip symbol, we set the variable `x` to 3:

```
var x;
x = 3;
```

On frame 1 of the circle clip symbol, we set the variable `y` to 4:

```
var y;
y = 4;
```

We place instances of those clips on frame 1, layer 1 of the main timeline of our movie and name our instances `square` and `circle`.

First question: If we attach the following code to frame 1 of the *main* movie timeline (upon which `square` and `circle` have been placed), what appears in the Output window? Here's the code:

```
trace(x);
trace(y);
```

Answer: Nothing appears in the Output window. The variables `x` and `y` are defined on the timelines of our movie clips, *not* our main timeline.

Variables attached to a movie clip timeline (like that of `square` or `circle`) have scope limited to that timeline. They are not directly accessible to scripts on other timelines, such as our main movie timeline.

Second question: If we were to place the `trace(x)` and `trace(y)` statements on frame 1 of our `square` movie clip instead of frame 1 of our main movie timeline, what would appear in the Output window? Here's the code:

Answer: The value of `x`, which is 3, and nothing else. The value of `x` is displayed because `x` is defined on the timeline of `square` and is therefore accessible to the *trace()* command, which also resides on that timeline. But the value of `y`, which is 4, doesn't appear in the Output window because `y` is defined in `circle`, which is a separate timeline.

You can now see why I said that ActionScript doesn't support true global variables. Global variables are variables that are accessible throughout an entire program, but in Flash, a variable attached to an individual timeline is directly accessible only to the scripts on that timeline. Since all variables in Flash are defined on timelines, no variable can be guaranteed to be directly accessible to *all* the scripts in a movie. Hence, no variable can legitimately be called *global*.

To prevent confusion, we refer to variables attached to timelines as *timeline variables* or *movie clip variables*. However, it is possible to simulate global variables using the *Object* class. To create a variable that is available on all timelines, use the following statement:

```
Object.prototype.myGlobalVariable = myValue;
```

For example:

```
Object.prototype.msg = "Hello world";
```

This technique (and the reason it works) is discussed under "The end of the inheritance chain" in Chapter 12, *Objects and Classes*.

Accessing Variables on Different Timelines

Even though variables on one timeline are not directly accessible to the scripts on other timelines, they are indirectly accessible. To create, retrieve, or assign a variable on a separate timeline, we use *dot syntax*, a standard notation common to object-oriented programming languages such as Java, C++, and JavaScript. Here's the generic dot syntax phrasing we use to address a variable on a separate timeline:

```
movieClipInstanceName.variableName
```

That is, we refer to a variable on another timeline using the name of the clip that contains the variable, followed by a dot, then the variable name itself. In our earlier scenario, for example, from the main timeline we would refer to the variable `x` in the `square` clip as:

```
square.x
```

Again, from the main timeline, we refer to the variable `y` in the `circle` clip as:

```
circle.y
```

We can use these references from our main movie timeline to assign and retrieve variables in `square` like this:

```
square.z = 5;       // Assign 5 to z in square
var mainZ;          // Create mainZ on the main timeline
mainZ = square.z;   // Assign mainZ the value of z in square
```

However, with just the *`clip.variable`* syntax alone, we can't refer to variables in `square` from our `circle` clip. If we were to put a reference to `square.x` on a frame in `circle`, the interpreter would try to find a clip called `square` *inside* of `circle`, but `square` lives on the main timeline. So, we need a mechanism that lets us refer to the timeline that contains the `square` clip (in this case, the main timeline) from the `circle` clip. That mechanism comes in the form of two special properties: `_root` and `_parent`.

The _root and _parent properties

The `_root` property is a direct reference to the main timeline of a movie. From any depth of nesting in a movie clip structure, we can always address variables on the main movie timeline using `_root`, like this:

```
_root.mainZ       // Access the variable mainZ on the main timeline
_root.firstName   // Access the variable firstName on the main timeline
```

We can even combine a reference to `_root` with references to movie clip instances, drilling down the nested structure of a movie in the process. For example, we can address the variable `x` inside the clip `square` that resides on the main movie timeline, as:

```
_root.square.x
```

That reference works from anywhere in our movie, no matter what the depth of clip nesting, because the reference starts at our main movie timeline, `_root`. Here's another nested example showing how to access the variable `area` in the instance `triangle` that resides on the timeline of the instance `shapes`:

```
_root.shapes.triangle.area
```

Any reference to a variable that starts with the `_root` keyword is called an *absolute reference* because it describes the location of our variable in relation to a fixed, immutable point in our movie: the main timeline.

There are times, however, when we want to refer to variables on other timelines without referring to the main timeline of a movie. To do so, we use the `_parent` property, which refers to the timeline upon which the current movie clip instance resides. For example, from code attached to a frame of the clip `square`, we can refer to variables on the timeline that *contains* `square` using this syntax:

```
_parent.myVariable
```

References that start with the keyword `_parent` are called *relative references* because they are resolved *relative* to the location of the clip in which they occur.

Returning to our earlier example, suppose we have a variable, `size`, defined on the main timeline of a movie. We place a clip instance named `shapes` on our main movie timeline, and on the `shapes` timeline we define the variable `color`. Also on the `shapes` timeline, we place a clip named `triangle`, as shown in Figure 2-1.

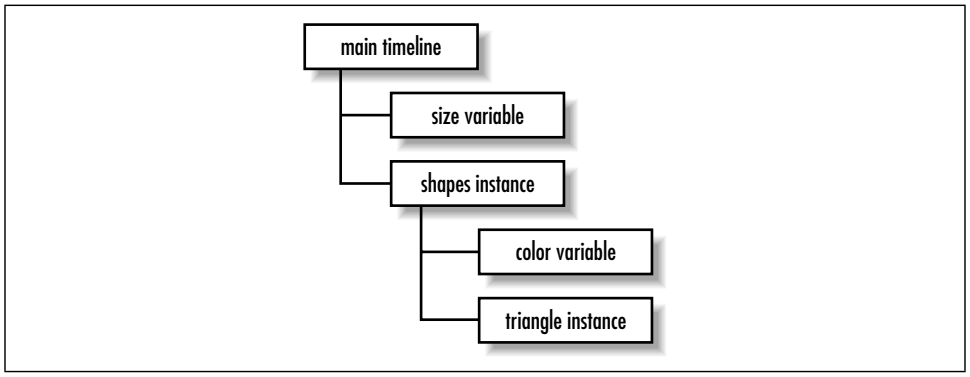

Figure 2-1. A sample movie clip hierarchy

To display the value of the variable `color` (which is in the `shapes` clip) from code attached to the timeline of `triangle`, we could use an absolute reference starting at the main timeline, like this:

```
trace(_root.shapes.color);
```

But that ties our code to the main movie timeline. To make our code more flexible, we could instead use the `_parent` property to create a relative reference, like this:

```
trace(_parent.color);
```

Our first approach (using `_root`) works from a top-down perspective; it starts at the main timeline and descends through the movie clip hierarchy until it reaches the `color` variable. The second approach (using `_parent`) works from a bottom-up perspective; it starts with the clip that contains the *trace()* statement (the `triangle` clip), then ascends one level up the clip structure where it finds the `color` variable.

We can use `_parent` twice in a row to ascend the hierarchy of clips and access our `size` variable on the main timeline. Here we attach some code to `triangle` that refers to `size` on the main movie timeline:

```
trace(_parent._parent.size);
```

Using the `_parent` property twice in succession takes us up two levels, which in this context brings us to the main timeline of the movie.

Your approach to variable addressing will depend on what you want to happen when you place instances of a movie clip symbol on various timelines. In our `triangle` example, if we wanted our reference to `color` to always point to `color` as defined in the `shapes` clip, then we would use the `_root` syntax, which gives us a fixed reference to `color` in `shapes`. But if we wanted our reference to `color` to refer to a different `color` variable, depending on which timeline held a given `triangle` instance, we would use the `_parent` syntax.

Accessing variables on different document levels

The `_root` property refers to the main movie timeline of the current level (i.e., the current document), but the Flash Player can accommodate multiple documents in its *document stack*. The main timeline of any movie loaded in the Player document stack may be referenced using `_level`*n*, where *n* is the level number on which the movie resides. Level numbers start with 0, such as `_level0`, `_level1`, `_level2`, `_level3`, and so on. For information on loading multiple movies, see Chapter 13, *Movie Clips*. Here are some examples showing multiple-level variable addressing:

```
_level1.firstName         // firstName on level1's main timeline
_level4.ball.area         // area in ball clip on level4's main timeline
_level0.guestBook.email   // email in guestBook clip on level0's timeline
```

When addressing variables across movie clip instance timelines using dot syntax, make sure that you have named your clip instances on the Stage and entered the names correctly when referring to them in your code. If your instances are not named, your code will not work even if it is otherwise syntactically correct. Unnamed instances and misspelled instance names are extremely common sources of problems.

Flash 4 versus Flash 5 variable access syntax

The Flash 4–style slash-colon constructions such as `/square:area` have been superseded by Flash 5's dot syntax, a much more convenient way to refer to variables and timelines. The old syntax is deprecated and no longer recommended. Table 2-1 shows equivalencies between Flash 4 and Flash 5 syntax when addressing variables. See Appendix C, *Backward Compatibility*, for other syntactical differences.

Table 2-1. Flash 4 Versus Flash 5 Variable Addressing Syntax

Flash 4 Syntax	Flash 5 Syntax	Refers to . . .
`/`	`_root`	Movie's main timeline
`/:x`	`_root.x`	Variable `x` on movie's main timeline
`/clip1:x`	`_root.clip1.x`	Variable `x` in instance `clip1` on movie's main timeline
`/clip1/clip2:x`	`_root.clip1.clip2.x`	Variable `x` in instance `clip2` within instance `clip1` within the main movie timeline
`../`	`_parent`	Timeline upon which the current clip resides (one level up from current clip timeline*)
`../:x`	`_parent.x`	Variable `x` on timeline upon which the current clip resides (one level up from current clip timeline)
`../../:x`	`_parent._parent.x`	Variable `x` on timeline that contains the clip that contains the current clip (two levels up from current clip timeline)
`clip1:x`	`clip1.x`	Variable `x` in instance `clip1`, where `clip1` resides on the current timeline
`clip1/clip2:x`	`clip1.clip2.x`	Variable `x` in instance `clip2`, where `clip2` resides within `clip1`, which, in turn, resides on current timeline
`_level1:x`	`_level1.x`	Variable `x` on the main timeline of a movie loaded onto level 1
`_level2:x`	`_level2.x`	Variable `x` on the main timeline of a movie loaded onto level 2

* The "current clip timeline" is the timeline that contains the code with the variable reference.

Movie Clip Variable Life Span

Earlier, we said that the scope of a variable answers two questions: (a) how long does the variable exist? and (b) from where in our code can we set or retrieve the variable's value? For movie clip variables, we now know the factors involved in answering the second question. But we skipped answering the first question. Let's return to it now with one final variable-coding scenario.

Scenario 5

Suppose we create a new movie with two keyframes. On frame 1, we place a clip instance, `ball`. On the `ball` timeline, we create a variable, `radius`. Frame 2 of our main timeline is blank (the `ball` instance is not present there).

From frame 1 of the main movie timeline, we can find out the value of `radius` using this code:

```
trace(ball.radius);
```

Now the question: If we move that line of code from frame 1 to frame 2 of the main timeline, what appears in the Output window when our movie plays?

Answer: Nothing appears. When the `ball` clip is removed from the main timeline on frame 2, all its variables are destroyed in the process.

Movie clip variables last only while the clip in which they reside is present on stage. Variables defined on the main timeline of a Flash document persist within each document but are lost if the document is unloaded from the Player (either via the *unloadMovie()* function or because another movie is loaded into the movie's level).

A variable's life span is important when scripting movies that contain movie clips placed across multiple frames on various timelines. Always make sure that any clip you're addressing is present on a timeline before you try to use the variables in that clip.

Local Variables

Movie clip variables are scoped to movie clips and persist as long as the movie clip on which they are defined exists. Sometimes, that's longer than we need them to live. For situations in which we need a variable only temporarily, ActionScript offers variables with *local* scope (i.e., local variables), which live for a much shorter time than normal movie clip variables.

Local variables are used in functions and older Flash 4–style subroutines. If you haven't worked with functions or subroutines before, you can skip the rest of this section and come back to it once you've read Chapter 9, *Functions*.

Functions often employ variables that are not needed outside the function. For example, suppose we have a function that displays all of the elements of a specified array:

```
function displayElements(theArray) {
   var counter = 0;
   while(counter < theArray.length) {
      trace("Element " + counter + ": " + theArray[counter]);
      counter++;
   }
}
```

The `counter` variable is required to display the array but has no use thereafter. We could leave it defined on the timeline, but that's bad form for two reasons: (a) if `counter` persists, it takes up memory during the rest of our movie; and (b) if `counter` is accessible outside our function, it may conflict with other variables named `counter`. We would, therefore, like `counter` to die after the *displayElements()* function has finished.

To cause `counter` to be automatically deleted at the end of our function, we define it as a *local variable*. Unlike movie clip variables, local variables are removed from memory (*deallocated*) automatically by the interpreter when the function that defines them finishes.

To specify that a variable should be local, declare it with the *var* keyword from inside your function, as in the preceding *displayElements()* example.

Take heed though; when placed *outside* of a function, the *var* statement creates a normal timeline variable, not a local variable. As shown in Example 2-4, the location of the *var* statement makes all the difference.

Variables within functions need not be local. We can create or change a movie clip variable from inside a function by omitting the *var* keyword. If we do not use the *var* keyword, but instead simply assign a value to a variable, Flash treats that variable as a nonlocal variable under some conditions. Consider this variable assignment inside a function:

```
function setHeight(){
   height = 10;
}
```

The effect of the statement `height = 10;` depends on whether `height` is a local variable or movie clip variable. If `height` is a previously declared local variable (which it is not in the example at hand), the statement `height = 10;` simply modifies the local variable's value. If there is no local variable named `height`, as is the case here, the interpreter creates a movie clip (nonlocal) variable named `height` and sets its value to 10. As a nonlocal variable, `height` persists even after the function finishes.

Example 2-4 demonstrates local and nonlocal variable usage.

Example 2-4. Local and Nonlocal Variables

```
var x = 5;                          // New nonlocal variable, x, is now 5
function variableDemo(){
   x = 10;                          // Nonlocal variable, x, is now 10
   y = 20;                          // New nonlocal variable, y, is now 20
   var z = 30;                      // New local variable, z, is now 30
   trace(x + "," + y + "," + z);    // Send variable values to Output window
}
variableDemo();   // Call our function. Displays: 10,20,30
trace(x);         // Displays: 10 (reassignment in our function was permanent)
trace(y);         // Displays: 20 (nonlocal variable, y, still exists)
trace(z);         // Displays nothing (local variable, z, has expired)
```

Note that it is possible (though confusing and ill-advised) to have both a local and a nonlocal variable that share the same name within a script but have different scopes. Example 2-5 shows such a case.

Example 2-5. Local and Nonlocal Variables with the Same Name

```
var myColor = "blue";
function hexRed(){
   var myColor = "#FF0000";
   return myColor;
}
trace(hexRed());   // Displays: #FF0000 (the local variable myColor)
trace(myColor);    // Displays: "blue" (setting the local variable,
                   // myColor, to #FF0000 did not affect the nonlocal version)
```

Local variables in subroutines

Although functions are the preferred mechanism for producing portable code modules, Flash 5 still supports Flash 4–style *subroutines*. In Flash 4, a subroutine could be created by attaching a block of code to a frame with a label. Later, the subroutine could be executed remotely via the *Call* action. But in Flash 4, any variable declared in a subroutine was nonlocal and persisted for the lifetime of the timeline on which it was defined. In Flash 5, you can create local variables in subroutines the same way we created them in functions—using the *var* statement. However, variables defined with *var* in a subroutine are created as local variables only when the subroutine is executed via the *Call* function. If the script on the subroutine frame is executed as a result of the playhead simply entering the frame, the *var* statement declares a normal timeline nonlocal variable. Regardless, the more modern functions and local function variables should be used instead of subroutines.

Some Applied Examples

We've had an awful lot of variable theory. How about showing some of these concepts in use? The following examples provide three variable-centric code samples. Refer to the comments for an explanation of the code.

Example 2-6 chooses a random destination for the playhead of a movie.

Example 2-6. Send the Playhead to a Random Frame on the Current Timeline

```
var randomFrame;            // Stores the randomly picked frame number
var numFrames;              // Stores the total number of frames on the timeline
numFrames = _totalframes;   // Assign _totalframes property to numFrames

// Pick a random frame
randomFrame = Math.floor(Math.random() * numFrames + 1);

gotoAndStop(randomFrame);   // Send playhead to chosen random frame
```

Example 2-7 determines the distance between two clips. A working version of this example is available from the online Code Depot.

Example 2-7. Calculate the Distance Between Two Movie Clips

```
var c;         // A convenient reference to the circle clip object
var s;         // A convenient reference to the square clip object
var deltaX;    // The horizontal distance between c and s
var deltaY;    // The vertical distance between c and s
var dist;      // The total distance between c and s

c = _root.circle;       // Get reference to the circle clip
s = _root.square;       // Get reference to the square clip
deltaX = c._x - s._x;   // Compute the horizontal distance between the clips
deltaY = c._y - s._y;   // Compute the vertical distance between the clips

// The distance is the root of (deltaX squared plus deltaY squared).
dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

// Tidy references are much more readable than the alternative:
dist = Math.sqrt(((_root.circle._x - _root.square._x) * (_root.circle._x -
_root.square._x)) + ((_root.circle._y - _root.square._y) * (_root.circle._y -
_root.square._y)));
```

Example 2-8 converts between Fahrenheit and Celsius. A working version is available in the online Code Depot.

Example 2-8. A Fahrenheit/Celsius Temperature Converter

```
var fahrenheit;           // Temperature in Fahrenheit
var celsius;              // Temperature in Celsius
var convertDirection;     // The system we are converting to.
                          // Legal values are "fahrenheit" and "celsius"
fahrenheit = 451;               // Set a Fahrenheit temperature
celsius = 20;                   // Set a Celsius temperature
convertDirection = "celsius";   // Convert to Celsius in this case
```

Example 2-8. A Fahrenheit/Celsius Temperature Converter (continued)

```
if (convertDirection == "fahrenheit") {
   result = (celsius * 1.8) + 32;   // Calculate the Celsius value.
   // Display the result
   trace (celsius + " degrees Celsius is " + result + " degrees Fahrenheit.");
} else if (convertDirection == "celsius") {
   result = (fahrenheit - 32) / 1.8;   // Calculate the Fahrenheit value.
   // Display the result
   trace (fahrenheit + " degrees Fahrenheit is " + result + " degrees Celsius.");
} else {
   trace ("Invalid conversion direction.");
}
```

Onward!

Now that we know all there is to know about storing information in variables, it's time we learn something more about the content that variables store: *data*. Over the next three chapters, we'll learn what data is, how it can be manipulated, and why it's an essential part of nearly everything we build with ActionScript.