CHAPTER 3

Data and Datatypes

Having worked with variable values in Chapter 2, you've already had a casual introduction to data, the information we manipulate in our scripts. In this chapter, we'll explore data in more depth, learning how ActionScript defines, categorizes, and stores data. We'll also explore how to create and classify data.

Data Versus Information

In the broadest sense, *data* is anything that can be stored by a computer, from words and numbers to images, video, and sound. All computer data is stored as a sequence of 1s and 0s, which you might recognize from high-tech marketing materials:

```
010101010101010110101011011010101010101010000010101010101011010101010
101010101010101011101010101010101010101010101010111110101010101010101
010101010101010101010101011101010101010101010101010101010101010101010
```

Data is information in its crude state—raw and meaningless. Semantics give information meaning. Consider, for example, the number 8008898969. As raw data it isn't very meaningful, but when we classify it semantically as the telephone number (800) 889-8969, the data becomes useful information.

This chapter shows how to add meaning to raw computer data so that it becomes human-comprehensible information.

Retaining Meaning with Datatypes

How do we store information as raw data without losing meaning? By categorizing our data and defining its datatype, we give it context that defines its meaning.

For example, suppose we have three numbers: 5155534, 5159592, and 4593030. By categorizing our data—as, say, a phone number, fax number, and parcel tracking number—the context (and, hence, the meaning) of our data is preserved. When categorized, each of the otherwise-nondescript seven-digit numbers becomes meaningful.

Programming languages use *datatypes* to provide rudimentary categories for data. For example, nearly all programming languages define datatypes to store and manipulate text (a.k.a. *strings*) and numbers. To distinguish between multiple numbers, we can use well-conceived variable names, such as `phoneNumber` and `faxNumber`. In more complex situations, we can create our own custom data categories with *objects* and *object classes*, as covered in Chapter 12. Before we think about making our own data categories, let's see which categories come built into ActionScript.

The ActionScript Datatypes

When programming, we may want to store a product name, a background color, or the number of stars to be placed in a night sky. We use the following ActionScript datatypes to store our data:

string
: For text sequences such as "`hi there`". A *string* is a series of characters (alphanumerics and punctuation).

number
: For numbers, such as 351 and 7.5. Numbers are used for counting and for mathematical equations.

boolean
: For logical decisions. With Boolean data, we can represent or record the status of some condition or the result of some comparison. Boolean data has only two legal values: `true` and `false`.

null and undefined
: For representing an *absence* of data, ActionScript provides two special data values: `null` and `undefined`. You can think of them as the only permissible values of the *null* and *undefined* datatypes.

array
: For lists of one or more pieces of data.

movieclip
: For manipulating movie clip instances.

object
: For arbitrary built-in or user-defined classes of data.

Every piece of data we store in ActionScript will fall into one of these categories. Before studying each datatype in Chapter 4, we'll consider the general issues that affect our use of all data.

Creating and Categorizing Data

There are two ways to create a new datum with ActionScript, and both methods require the use of *expressions*—phrases of code that represent data in scripts.

A literal expression (or *literal* for short) is a series of letters, numbers, and punctuation that *is* the datum. A data literal is a verbatim description of data in a program's source code. This contrasts with a *variable*, which is merely a container that holds a datum. Each datatype defines its own rules for the creation of literals. For example, string literals are enclosed in quotes, whereas numeric literals are not. Here are some examples of literals:

```
"loading...please wait"  // A string literal
1.51                     // A numeric literal
["jane", "jonathan"]     // An array literal
{x: 10, y: 15}           // An object literal
```

Note that movie clips cannot be represented by literals but are referred to by instance names.

We can also generate data programmatically with a *complex expression*. Complex expressions are phrases of code with a value that must be calculated or computed, not taken literally. The calculated value is the datum being represented. For example, each of these complex expressions results in a single datum:

```
1999 + 1       // Yields the numeric datum 2000
"1999" + "1"   // Yields the string datum "19991"
"hi " + "ma!"  // Yields the string datum "hi ma!"
firstName      // Yields the value of the variable firstName
_currentframe  // Yields the frame number of the playhead's current position
new Date()     // Yields a new Date object with the current date and time
```

Notice that an individual literal expression, such as 1999 or 1, can be a valid part of a larger complex expression, as in *1999 + 1*.

Whether we use a literal expression or a complex expression to create data, we must store every datum that we want to use later. The result of the expression `"hi" + "ma!"` is lost unless we store it, say, in a variable. For example:

```
// This datum is fleeting and dies immediately after it's created
"hi " + "ma";
// This datum is stored in a variable and can be
// accessed later via the variable welcomeMessage
var welcomeMessage = "hi " + "ma!";
```

How do we categorize data into the appropriate type? That is, how do we specify that a datum is a number, a string, an array, or whatever? In most cases, we don't categorize new data ourselves; the ActionScript interpreter automatically assigns or infers each datum's type based on a set of internal rules.

Automatic Literal Typing

The interpreter infers a literal datum's type by examining its syntax, as explained in the comments in the following code fragment:

```
"animal"            // Quotation marks identify "animal" as a string
1.35                // If it contains only integers and a decimal point,
                    // it is a number
```

```
true                 // Special keyword true identifies this as a Boolean
null                 // Special keyword null identifies this as the null type
undefined            // Special keyword undefined identifies the undefined type
["hello", 2, true]   // Square brackets and values separated by commas
                     // indicate that this is an array
{x: 234, y: 456}     // Curly braces and property name/value pairs separated
                     // by commas indicate that this is an object
```

As you can see, using correct syntax with data literals is extremely important. Incorrect syntax may cause an error or result in the misinterpretation of a datum's content. For example:

```
animal   // Missing quotes--animal is interpreted as a variable,
         // not a string of text
"1.35"   // Numbers in quotes are treated as strings, not numbers
1. 35    // Space before the 3 causes an error
"animal  // Missing closing quotation mark causes an error
```

Automatic Complex Expression Typing

The interpreter computes an expression's value in order to determine its datatype. Consider this example:

```
pointerX = _xmouse;
```

Because `_xmouse` stores the location of the mouse pointer as a number, the type of the expression `_xmouse` will always be a number; so, the variable `pointerX` also becomes a number.

Usually, the datatype that is automatically determined by the interpreter matches what we expect and want. However, some ambiguous cases require us to understand the rules that the interpreter uses to determine an expression's datatype (see Example 2-2 and Example 2-3). Consider the following expression:

```
"1" + 2;
```

The operand on the left of the + is a string ("1"), but the operand on the right is a number (2). The + operator works on both numbers (addition) and strings (concatenation). Should the value of the expression `"1" + 2` be the number 3 or the string "12"? To resolve the ambiguity, the interpreter relies on a fixed rule: the plus operator (+) always favors strings over numbers, so the expression `"1" + 2` evaluates to the string "12", not the number 3. This rule is arbitrary, but it provides a consistent way to interpret the code. The rule was chosen with typical uses of the plus operator in mind: if one of the operands is a string, it's likely that we want to concatenate the operands, not add them numerically, as in this case:

```
trace ("The value of x is: " + x);
```

Combining disparate types of data or using a datum in a context that does not match the expected datatype causes ambiguity. This forces the interpreter to perform an automatic datatype *conversion* according to arbitrary, but predictable, rules. Let's examine the cases in which automatic conversions will occur and the results of converting a datum from one type to another.

Datatype Conversion

Take a closer look at the example from the previous section. In that example, each datum—“1” and 2—belonged to its own datatype; the first was a string and the second was a number. We saw that the interpreter joined the two values together to form the string “12”. Note that the interpreter first had to *convert* the *number* 2 into the *string* “2”. Only after that automatic conversion was performed could the value “2” be joined (concatenated) to the string “1”.

Datatype conversion simply means changing the type of a datum. Not all datatype conversions are automatic; we may also change a datum’s type explicitly in order to override the default datatype conversion that ActionScript would otherwise perform. Explicit conversion is known as *typecasting*, or simply *casting*.

Automatic Type Conversion

Whenever we use a value in a context that does not match the expected datatype, the interpreter attempts a conversion. That is, if the interpreter expects data of type A, and we provide data of type B, the interpreter will attempt to convert our type B data into type A data. For example, in the following code we use the string “Flash” as the right-hand operand of the subtraction operator. Since only numbers may be used with the subtraction operator, the interpreter attempts to convert the string “Flash” into a number:

```
999 - "Flash";
```

Of course, the string “Flash” can’t be successfully converted into a legitimate number, so it is converted into the special numeric data value `NaN` (i.e., Not-a-Number). `NaN` is a legal value of the *number* datatype, intended specifically to handle such a situation. With “Flash” converted to `NaN`, our expression ends up looking like this to the interpreter (though we never see this interim step):

```
999 - NaN;
```

Both operands of the subtraction operator are now numbers, so the operation can proceed: `999 - NaN` yields the value `NaN`, which is the final value of our expression.

An expression that yields the numeric value `NaN` isn’t particularly useful; most conversions have more functional results. For example, if a string contains only numeric characters, it can be converted into a useful number. The expression:

```
999 - "9";  // The number 999 minus the string "9"
```

is interpreted as:

```
999 - 9;    // The number 999 minus the number 9
```

which yields the value 990 when the expression is resolved. Automatic conversion is most common with the plus operator, the equality operator, the comparison operators, and in conditional or loop statements. In order to be sure of the result of any expression that involves automatic conversion, we have to answer three questions: (a) what is the expected datatype of the current context? (b) what happens when an unexpected datatype is supplied in that context? and (c) when conversion occurs, what is the resulting value?

To answer the first and second questions, we need to consult the appropriate topics elsewhere in this book (e.g., to determine what datatype is expected in a conditional statement, see Chapter 7).

The next three tables, which list the rules of automatic conversion, answer the third question, "When conversion occurs, what is the resulting value?" Table 3-1 shows the results of converting each datatype to a number.

Table 3-1. Converting to a number

Original data	Result after conversion
`undefined`	`0`
`null`	`0`
Boolean	`1` if the original value is `true`; `0` if the original value is `false`
Numeric string	Equivalent numeric value if string is composed only of base-10 numbers, whitespace, exponent, decimal point, plus sign, or minus sign (e.g., "`-1.485e2`")
Other strings	Empty strings, nonnumeric strings, including strings starting with "`x`", "`0x`", or "`FF`", convert to `NaN`
"`Infinity`"	`Infinity`
"`-Infinity`"	`-Infinity`
"`NaN`"	`NaN`
Array	`NaN`
Object	The return value of the object's *valueOf()* method
Movieclip	`NaN`

Table 3-2 shows the results of converting each datatype to a string.

Table 3-2. Converting to a string

Original data	Result after conversion
`undefined`	"" (the empty string)
`null`	"`null`"
Boolean	"`true`" if the original value was `true`; "`false`" if the original value was `false`.
`NaN`	"`NaN`"
`0`	"`0`"
`Infinity`	"`Infinity`"
`-Infinity`	"`-Infinity`"

Table 3-2. Converting to a string (continued)

Original data	Result after conversion
Other numeric value	String equivalent of the number. For example, `944.345` becomes "`944.345`".
Array	A comma-separated list of element values.
Object	The value that results from calling *toString()* on the object. By default, the *toString()* method of an object returns "`[object Object]`". The *toString()* method can be customized to return a more useful result (e.g., *toString()* of a *Date* object returns: "`Sun May 14 11:38:10 EDT 2000`").
Movieclip	The path to the movie clip instance, given in absolute terms starting with the document level in the Player. For example, "`_level0.ball`".

Table 3-3 shows the results of converting each datatype to a Boolean.

Table 3-3. Converting to a Boolean

Original data	Result after conversion
`undefined`	`false`
`null`	`false`
`NaN`	`false`
`0`	`false`
`Infinity`	`true`
`-Infinity`	`true`
Other numeric value	`true`
Nonempty string	`true` if the string can be converted to a valid nonzero number, `false` if not; in ECMA-262, a nonempty string always converts to `true` (Flash diverges from the ECMA standard to maintain compatibility with Flash 4)
Empty string ("")	`false`
Array	`true`
Object	`true`
Movieclip	`true`

Explicit Type Conversion

If the automatic (implicit) type-conversion rules do not suit our purpose, we can manually (explicitly) change a datum's type. When we take matters into our own hands, we must remember that the rules listed in the preceding tables still apply.

Converting to a string with the toString( ) method

We can invoke the *toString()*method to convert any datum to a string. For example:

```
x.toString( );      // Get the string value of the variable x.
(523).toString( );  // Returns "523". Note that we use parentheses
                    // so that the "." isn't treated as a decimal point.
```

When we invoke the *toString()* method on a number, we may also provide a numeric argument indicating the base of the number system in which we'd like the converted string to be represented. This provides a handy means of switching between hexadecimal, decimal, and octal numbers. For example:

```
var myColor = 255;
var hexColor = myColor.toString(16);  // Sets hexColor to "ff"
```

Converting to a string with the String( ) function

The *String()* function has the same result as the *toString()* method, but it uses a different grammar:

```
String(x);    // Convert x to a string
String(523);  // Convert 523 to the string "523"
```

Don't confuse the global *String()* function with the built-in class constructor of the same name. Both are described in the *Language Reference*.

Converting to a string with empty string concatenation

Because the plus operator (+) favors strings in its automatic conversion rules, concatenating the empty string ("") with any datum converts that datum to a string.

```
// Convert x to a string.
x + "";
// Here we check the character position of the number 2 in 523. We first
// concatenate 523 and "", before invoking a String method on the converted value.
trace((523 + "").indexOf(2));
```

Converting to a number with the Number( ) function

Just as the *String()* function converts data to the *string* type, the *Number()* function converts its argument to the *number* type. When conversion to a real number is impossible or illogical, the *Number()* function returns a special numeric value as described in Table 3-1. Here are some examples:

```
Number(age);     // Yields the value of age converted to a number
Number("29");    // Yields the number 29
Number("sara");  // Yields NaN
```

Don't confuse the global *Number()* function with the built-in class constructor of the same name. Both are described in the *Language Reference*.

Because user input in on-screen text fields always belong to the string type, it's necessary to convert text fields to numbers when performing mathematical calculations. For example, if we want to find the sum of the text fields `price1_txt` and `price2_txt`, we use:

```
totalCost = Number(price1_txt.text) + Number(price2_txt.text);
```

Otherwise, `price1_txt` and `price2_txt` will be concatenated as strings, not added as numbers. For more information on text fields, see *TextField* in the *Language Reference*.

Converting to a number by subtracting zero

To trick the interpreter into converting a datum to a number, we can subtract zero from that datum. Again, the conversion follows the rules described in Table 3-1:

```
"953" - 0      // Yields 953
"molly" - 0    // Yields NaN
x - 0          // Yields the value of x converted to a number
```

Converting to a number using the parseInt( ) and parseFloat( ) functions

The *parseInt()* and *parseFloat()* functions convert a string containing numbers and letters into a number. The *parseInt()* function extracts the first integer that appears in a string, provided that the string's first nonblank character is a legal numeric character. Otherwise, *parseInt()* yields NaN. The number extracted via *parseInt()* starts with the first nonblank character in the string and ends with the character before either the first nonnumeric character or the first occurrence of a decimal point.

Here are some *parseInt()* examples:

```
parseInt("1a")                // Yields 1
parseInt("1.3a"               // Yields 1
parseInt("     1a")           // Yields 1
parseInt("I am 14 years old") // Yields NaN (the first nonblank
                              // character is not a number)
parseInt("14 years old")      // Yields 14

// Convert decimal to hexadecimal.
(255).toString(16);           // Yields: ff
// Convert hexadecimal to decimal.
parseInt("0xFF");             // Yields 255
```

The *parseFloat()* function returns the first floating-point number that appears in a string, provided that the string's first nonblank character is a valid numeric character. (A floating-point number is a positive or negative number that contains a decimal value, such as -10.5 or 345.678.) Like *parseInt()*, *parseFloat()* yields the special numeric value NaN if the string's first nonblank character is not a valid numeric character. The number extracted by *parseFloat()* is the numeric conversion of the series of characters that starts with the first nonblank character in the string and ends with the character before the first nonnumeric character (any character other than +, -, 0–9, a decimal point, or an *e* or *E* when used for exponential notation).

Here are some *parseFloat()* examples:

```
parseFloat("1.3a");            // Extracts 1.3
parseFloat("2.75 years old")   // Extracts 2.75
parseFloat("1nce upon a time") // Extracts 1
parseFloat("I'm 3.5 feet tall") // Yields NaN
```

For more information on *parseInt()* and *parseFloat()*—including how to specify a *radix* to convert between number systems—see the *Language Reference*.

Converting to a Boolean

When we want to convert a datum to a Boolean, we can use the global *Boolean()* function, which uses similar syntax to the *String()* and *Number()* functions. For example:

```
Boolean(5);  // The result is true
Boolean(x);  // Converts value of x to a Boolean
```

Don't confuse the global *Boolean()* function with the built-in class constructor of the same name. Both are described in the *Language Reference*.

Conversion Duration

All type conversions performed on variables, array elements, and object properties are temporary unless the conversion happens as part of an assignment. Here we see a temporary conversion:

```
var x = "10";     // x is a string.
y = x - 5;        // y is now 5; x's value was temporarily converted to a number.
trace(typeof x);  // Displays: "string"; the conversion was temporary because
                  // it occurred incidentally while evaluating an expression.
```

Here we see a permanent conversion that is the result of an assignment:

```
x = "10";         // x is a string.
x = x - 5;        // x is converted permanently to a number.
trace(typeof x);  // Displays: "number"; the conversion was permanent because
                  // it occurred as part of an assignment.
```

Determining the Type of an Existing Datum

To determine what kind of data is held in a given expression before, say, proceeding with a section of code, we use the *typeof* operator, as follows:

```
typeof expression;
```

The *typeof* operator returns a string telling us the datatype of *`expression`*, according to Table 3-4.

Table 3-4. Return values of typeof

Original datatype	typeof return value
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Array	"object"
null	"null"
Movieclip	"movieclip"

Table 3-4. Return values of typeof (continued)

Original datatype	typeof return value
Function	"function"
undefined	"undefined"

Here are a few examples:

```
trace(typeof "game over");    // Displays: "string" in the Output window
var x = 5;
trace(typeof x);              // Displays: "number"
var now = new Date();
trace(typeof now);            // Displays: "object"
```

As shown in Example 3-1, when combined with a *for-in* statement, *typeof* provides a handy way to find all the movie clip instances on a timeline. Once the clips are identified, we can assign them to an array for programmatic handling. (If you can't follow all of Example 3-1, revisit it after completing Part I.)

Example 3-1. Populating an array with dynamically identified movie clips

```
// Create an array in which to store the clips.
var childClips = new Array( );

// Check all the properties of the main timeline.
for (prop in _root) {
  // If the current property is a movie clip...
  if (typeof _root[prop] == "movieclip") {
    // ...add it to the clips array.
    childClips.push(_root[prop]);
  }
}

// Now that our array is populated, we can use it to manipulate the clips it contains.
childClips[0]._x = 0;  // Place the first clip on the left of the Stage.
childClips[1]._y = 0;  // Place the second clip at the top of the Stage.
```

Primitive Data Versus Composite Data

So far we've been working mostly with numbers and strings, which are the most common *primitive* datatypes. Primitive datatypes are the basic units of a language; each primitive value represents a single datum (as opposed to an array of multiple items) and holds that datum directly, rather than holding its address elsewhere in memory.

ActionScript supports these primitive datatypes: *number*, *string*, *boolean*, *undefined*, and *null*. ActionScript does not have a separate single-character datatype (e.g., *char*) as found in C/C++ (strings are a primitive datatype in ActionScript, and not arrays of chars as they are in C/C++).

Primitive datatypes are, as their name suggests, simple. They can hold text messages, frame numbers, secret passwords, and so on, but they don't readily accommodate higher levels of complexity. For more elaborate data handling—such as simulating the physics of a dozen bouncing balls or managing a quiz with 500 questions and answers—we turn to *composite* datatypes. Using composite data, we can manage multiple pieces of related data as a single datum.

ActionScript supports the following composite datatypes: *array*, *object*, and *movieclip*. Technically, functions are a type of object and are therefore considered composite data, but we rarely manipulate them as such. See Chapter 9 for more about functions as a datatype.

Whereas a single number is a primitive datum, a list (i.e., an *array*) of multiple numbers is a composite datum. Here's a practical example of how composite datatypes are useful: suppose we want to store the name of a customer named Derek. We can create a variable that stores Derek's name as a primitive value, like this:

```
var custName = "Derek";
```

However, this approach gets pretty cumbersome once we add more customers. We're forced to use sequentially named variables to keep track of our customers—`cust1Name`, `cust2Name`, `cust3Name`, and so on. Yuck! But if we use an array, we can store our information much more efficiently:

```
customers = ["Derek", "James", "Joe"];
```

Now that's nice and tidy. We'll learn much more about composite datatypes in the coming chapters.

Copying, Comparing, and Passing Data

There are three fundamental ways to manipulate data; we can *copy* it (e.g., assign the value of variable `x` to variable `y`), *compare* it (e.g., check whether `x` equals `y`), and *pass* it (e.g., supply a variable to a function as an argument). Primitive data values are copied, compared, and passed quite differently than composite data. When primitive data is copied to a variable, that variable gets its own unique and private copy of the data, stored separately in memory. Hence, the following lines of code cause the string "Dave" to be stored twice in memory, once in the memory location reserved for `name1` and again in the location reserved for `name2`:

```
name1 = "Dave";
name2 = name1;
```

We say that primitive data is copied *by value* because the data's literal value is stored in the memory location allotted to the variable. In contrast, when composite data is copied to a variable, only a *reference* to the data (and not the actual data) is stored in the variable's memory slot. That reference tells the interpreter where the actual data is kept (i.e., its address in memory). When a variable that contains composite data is copied to another variable, it is the *reference* (often called a *pointer*) and not the data itself that is copied. Hence, composite data is said to be copied *by reference*.

This makes good design sense, because it would be grossly inefficient to duplicate large arrays and other composite datatypes, but it has important consequences for our code. When multiple variables are assigned the same piece of composite data as their value, each variable does *not* store a unique copy of the data (as it would if the data were primitive). Rather, multiple variables can point to one copy of the composite data. If the value of the data changes, all the variables that point to it reflect the updated value.

Let's see how this affects a practical application. When two variables refer to the same primitive data, each variable gets its own copy of the data. Here we assign the value 12 to the variable x:

```
var x = 12;
```

Now let's assign the value of x to a new variable, y:

```
var y = x;
```

As you can guess, y is now equal to 12. But y has its own copy of the value 12, distinct from the copy in x. If we change the value of x, the value of y is unaffected:

```
x = 15;
trace(x); // Displays: 15
trace(y); // Displays: 12
```

The value of y did not change when x changed because when we assigned x to y, y received its own copy of the number 12 (i.e., the *primitive* datum contained by x).

Now let's try the same thing with *composite* data. We'll create a new array with three elements and then assign that array to the variable x:

```
var x = ["first element", 234, 18.5];
```

Now, just as we did before, we'll assign the value of x to y:

```
var y = x;
```

The value of y is now the same as the value of x. But what is the value of x ? Remember that because x refers to an array, which is a composite datum, the value of x is not literally the array ["first element", 234, 18.5] but merely a reference to that datum. Hence, when we assign x to y, what's copied to y is not the array itself but the reference contained in x that points to the array. So, both x and y point to the same array, stored somewhere in memory.

If we change the array through the variable x, like this:

```
x[0] = "1st element";
```

the change is also reflected in y:

```
trace(y[0]);  // Displays: "1st element"
```

Similarly, if we modify the array through y, the change can be seen via x:

```
y[1] = "second element";
trace (x[1]);  // Displays: "second element"
```

To break the association, use the *slice()* function to create an entirely new array:

```
var x = ["first element", 234, 18.5];
// Copy each element of x to a new array stored in y
var y = x.slice(0);
y[0] = "hi there";
trace(x[0]);  // Displays: "first element" (not "hi there")
trace(y[0]);  // Displays: "hi there" (not "first element")
```

Let's extend our example to see how primitive and composite data values are *compared*. Here we assign x and y an identical primitive value; then we compare the two variables:

```
x = 10;
y = 10;
trace(x == y);  // Displays: true
```

Because x and y contain primitive data, they are compared by value. In a value-based comparison, data is compared literally. The number 10 in x is considered equal to the number 10 in y because the numbers are made up of the same bytes.

Now, let's assign x and y identical versions of the same composite data and compare the two variables again:

```
x = [10, "hi", 5];
y = [10, "hi", 5];
trace(x == y);  // Displays: false
```

This time, x and y contain composite data, so they are compared by reference. The arrays we assigned to x and y have the same byte values, but the variables x and y are not equal because they do not store a reference to the same composite datum. However, watch what happens when we copy the reference in y to x:

```
x = y;
trace(x == y);  // Displays: true
```

Now that the references are the same, the values are considered equal. Thus, the result of the comparison depends on the references in the variables, not the actual byte values of the arrays to which they point.

Primitive and composite data are also treated differently when passed to functions, as discussed under "Passing Parameters by Value Versus by Reference" in Chapter 9. Most notably, when a primitive variable is passed as an argument to a function, any changes to the datum within the function are not reflected in the original variable. However, when passing a composite variable, changes within the function do affect the original variable. That is, if you pass an integer variable x to a function, changes to the parameter within the function don't affect its original value in the calling routine. But if you pass an array y to a function, any changes to that array within the function will alter the original value of y outside the function (because changes to the array affect the data to which y points).

Onward!

We've introduced data in ActionScript, and we're ready for deeper study. In Chapter 4, we'll study the *number*, *string*, *boolean*, *undefined*, and *null* datatypes. In Chapter 5, we'll explore how to manipulate data using operators. In later chapters, we'll study the complex datatypes, such as *movieclips*, *arrays* and *objects*.