CHAPTER 2

Data and Decisions

Data and decision making are fundamental to every "intelligent" program. We'll begin this chapter by looking at how JavaScript understands, or represents, data. This is important because JavaScript works with a number of **data types** and manipulates data according to its data type. You can generate unexpected results by mismatching data of different types. We'll look at some of the more common data type problems, and you'll see how to convert one type of data to another.

We'll also be working with **conditional statements** and **loops**: two of the most valuable tools for decision making. In order to make decisions in a computer language, we need to let the program know what should happen in response to certain conditions, which is where conditional statements come in. Loops, on the other hand, simply allow you to repeat an action until a specified circumstance is met. For example, you might want to loop through each input box in a form and check the information it contains is valid.

I'll be covering a lot of different facets of JavaScript in this chapter:

- Classifying and manipulating information in JavaScript: data types and data operators
- Variables
- Converting data types
- Introducing data objects: `String`, `Date`, and `Math` objects
- Arrays: storing ordered sets of data like the items in a shopping basket
- Decision making with conditional statements, loops, and data evaluation

---

Note The examples in this chapter are kept as simple as possible and therefore use `document.write()` as a feedback mechanism for you to see the results. You'll learn in later chapters about other methods for doing this that are more modern and versatile.

Data, Data Types, and Data Operators

Data is used to store information, and, in order to do that more effectively, JavaScript needs to have each piece of data assigned a **type**. This type stipulates what can or cannot be done with the data. For example, one of the JavaScript data types is **number**, which allows you to perform certain calculations on the data that it holds.

The three most basic data types that store data in JavaScript are

- **String**: A series of characters, for example, "`some characters`"
- **Number**: A number, including floating point numbers
- **Boolean**: Can contain a true or false value

These are sometimes referred to as **primitive** data types in that they store only single values. There are two slightly different primitive data types as well. These don't store information, but instead warn us about a particular situation:

- **Null**: Indicates that there is no data.
- **Undefined**: Indicates that something has not been defined and given a value. This is important when you're working with variables.

We'll be working extensively with these data types throughout the chapter.

The String Data Type

The JavaScript interpreter expects string data to be enclosed within single or double quotation marks (known as **delimiters**). This script, for example, will write `some characters` onto the page:

```
<html>
  <body>
    <script type="text/javascript">
      document.write( "some characters" );
    </script>
  </body>
</html>
```

The quotation marks won't be written out to the page because they are not part of the string; they simply tell JavaScript where the string starts and ends. We could just as easily have used single quotation marks:

```
<html>
  <body>
    <script type="text/javascript">
      document.write( 'some characters' );
    </script>
  </body>
</html>
```

Both methods are fine, just as long as you close the string the same way you opened it and don't try to delimit it like this:

```
document.write( 'some characters" );
document.write( "some characters' );
```

Of course, you might want to use a single or double quotation mark inside the string itself, in which case you need to use a distinct delimiter. If you used double quotation marks, the instructions will be interpreted as you intended:

```
document.write( "Paul's characters " );
```

But if you used single quotations marks, they won't be:

```
document.write( 'Paul's characters' );
```

This will give you a syntax error because the JavaScript interpreter thinks the string ends after the *l* in *Paul* and doesn't understand what is happening afterwards.

■ **Note** JavaScript syntax, like English syntax, is a set of rules that makes the language "intelligible." Just as a syntax error in English can render a sentence meaningless, a syntax error in JavaScript can render the instruction meaningless.

You can avoid creating JavaScript syntax errors like this one by using single quotation marks to delimit any string containing double quotes and vice versa:

```
document.write( "Paul's numbers are 123" );
document.write( 'some "characters"' );
```

If, on the other hand, you wanted to use both single and double quotation marks in your string, you need to use something called an **escape sequence**. In fact, it's better coding practice to use escape sequences instead of the quotation marks we've been using so far, because they make your code easier to read.

Escape Sequences

Escape sequences are also useful for situations where you want to use characters that can't be typed using a keyboard (like the symbol for the Japanese yen, ¥, on a Western keyboard). Table 2-1 lists some of the most commonly used escape sequences.

Table 2-1. *Common Escape Sequences*

Escape Sequences	Character Represented
`\b`	Backspace.
`\f`	Form feed.
`\n`	Newline.
`\r`	Carriage return.
`\t`	Tab.
`\'`	Single quote.
`\"`	Double quote.
`\\`	Backslash.
`\xNN`	*NN* is a hexadecimal number that identifies a character in the Latin-1 character set (the Latin-1 character is the norm for English-speaking countries).
`\uDDDD`	*DDDD* is a hexadecimal number identifying a Unicode character.

Let's amend this string, which causes a syntax error:

```
document.write( 'Paul's characters' );
```

so that it uses the escape sequence (`\'`) and is correctly interpreted:

```
document.write( 'Paul\'s characters' );
```

The escape sequence tells the JavaScript interpreter that the single quotation mark belongs to the string itself and isn't a delimiter.

ASCII is a character encoding method that uses values from 0 to 254. As an alternative, we could specify characters using the ASCII value in hexadecimal with the `\xNN` escape `sequence`. The letter *C* is 67 in decimal and 43 in hex, so we could write that to the page using the escape sequence like this:

```
document.write( "\x43" );
```

The `\uDDDD` escape sequence works in much the same way but uses the Unicode character encoding method, which has 65,535 characters. As the first few hundred ASCII and Unicode character sets are similar, you can write out the letter *C* using this escape sequence as follows:

```
document.write( '\u0043' );
```

ASCII and Unicode information can get quite detailed, so the best place to look for information is on the Web. For Unicode, try `http://www.unicode.org`.

Operators

JavaScript has a number of operators that you can use to manipulate the data in your programs; you'll probably recognize them from math. Table 2-2 presents some of the most commonly used operators.

Table 2-2. *JavaScript Operators*

Operator	What It Does
+	Adds two numbers together or concatenates two strings.
-	Subtracts the second number from the first.
*	Multiplies two numbers.
/	Divides the first number by the second.
%	Finds the modulus—the reminder of a division. For example, 98 % 10 = 8.
--	Decreases the number by 1: only useful with variables, which we'll see at work later.
++	Increases the number by 1: only useful with variables, which we'll see at work later.

Here they are in use:

```
<html>
  <body>
    <script type="text/javascript">
      document.write( 1 - 1 );
      document.write( "<br />" );
      document.write( 1 + 1 );
      document.write( "<br />" );
      document.write( 2 * 2 );
      document.write( "<br />" );
      document.write( 12 / 2 );
      document.write( "<br />" );
      document.write( 1 + 2 * 3 );
      document.write( "<br />" );
      document.write( 98 % 10 );
    </script>
  </body>
</html>
```

You should get this output:

```
0
2
4
6
7
8
```

JavaScript, just like math, gives some operators precedence. Multiplication takes a higher precedence than addition, so the calculation 1 + 2 * 3 is carried out like this:

2 * 3 = 6

6 + 1 = 7

All operators have an order of precedence. Multiplication, division, and modulus have equal precedence, so where they all appear in an equation the sum will be calculated from left to right. Try this calculation:

2 * 10 / 5%3

The result is 1, because the calculation simply reads from left to right:

2 * 10 = 20

20 / 5 = 4

4%3 = 1

Addition and subtraction also have equal precedence.

You can use parentheses to give part of a calculation higher precedence. For example, you could add 1 to 1 and then multiply by 5 like this:

(1 + 1) * 5

The result will then be 10, but without the parentheses it would have been 6. In fact, it's a good idea to use parentheses even when they're not essential because they help make the order of the execution clear.

If you use more than one set of parentheses, JavaScript will simply work from left to right or, if you have inner parentheses, from the inside out:

```
document.write( ( 1 + 1 ) * 5 * ( 2 + 3 ) );
```

This is how the calculations for the preceding are performed:

(1 + 1) = 2

(2 + 3) = 5

2 * 5 = 10

10 * 5 = 50

As we've seen, JavaScript's addition operator adds the values. What it actually does with the two values depends on the data type that you're using. For example, if you're working with two numbers that have been stored as the number data type, the + operator will add them together. However, if one of the data types you're working with is a string (as indicated by the delimiters), the two values will be concatenated. Try this:

```
<html>
  <body>
    <script type="text/javascript">
      document.write( 'Java' + 'Script' );
      document.write( 1 + 1 );
      document.write( 1 + '1' );
    </script>
  </body>
</html>
```

Being able to use the addition operator with strings can be handy (and is called the **concatenation operator** in this case), but it can also generate unexpected results if one of the values you're working with happens to be of a different data type from the one you were expecting. We'll be looking at some examples like this, and resolving them, later on.

It's less of a problem if you're working with **literal** values as we have been doing so far. However, much of the data you'll be working with in your programs will be entered by the user, or generated by the script, so you won't know in advance exactly what values you're going to be working with. This is where **variables** come in. Variables are placeholders for data in your script, and they're central to JavaScript.

JavaScript Variables

JavaScript is probably the most forgiving language when it comes to variables. You don't need to define what a variable is before you can use it, and you can change the type of a variable any time in the script. However, to ease maintenance and keep up a stricter coding syntax, it is a good idea to declare variables explicitly at the beginning of your script or—in the case of local variables—your function.

We declare a variable by giving it a unique name and using the `var` keyword.

Variable names have to start with a letter of the alphabet or with an underscore, while the rest of the name can be made up only of numbers, letters, the dollar sign (`$`), and underscore characters. Do not use any other characters.

■ **Note** Like most things in JavaScript, variable names are case sensitive: `thisVariable` and `ThisVariable` are different variables. Be very careful about naming your variables; you can run into all sorts of trouble if you don't name them consistently. To that end, most programmers use **camel notation**: the name of the variable begins with a lowercase letter while subsequent words are capitalized and run in without spaces. Thus the name `thisVariable`.

Always give your variables meaningful names. In the next example we'll build, we're going to write an exchange rate conversion program, so we'll use variable names like `euroToDollarRate` and `dollarToPound`. There are two advantages to naming variables descriptively: it's easier to remember what the code is doing if you come back to it at a later date, and it's easier for someone new to the code to see what's going on. Code readability and layout are very important to the development of web pages. It makes it quicker and easier to spot errors and debug them, and to amend the code as you want to.

■ **Note** While it is not technically necessary, variable declarations should begin with the keyword `var`. Not using it could have implications, which you will see as we progress.

With all that said, let's start declaring variables. We can declare a variable without initializing it (giving it a value):

```
var myVariable;
```

Then it's ready and waiting for when we have a value. This is useful for variables that will hold user input.

We can also declare and initialize the variable at the same time:

```
var myVariable = "A String";
var anotherVariable = 123;
```

Or we can declare and initialize a variable by assigning it the return value of the `prompt()` function or the sum of a calculation:

```
var eurosToConvert = prompt( "How many Euros do you wish to➥
 convert", "" );
var dollars = eurosToConvert * euroToDollarRate;
```

The `prompt()` function is a JavaScript function that asks the user to enter a value and then returns it to the code. Here we're assigning the value entered to the variable `eurosToConvert`.

Initializing your variables is a very good idea, especially if you can give them a default value that's useful to the application. Even initializing a variable to an empty string can be a good idea, because you can check back on it without bringing up the error messages that would have popped up if it didn't have a value.

Let's look at how variables can improve both the readability of your code and its functionality. Here's a block of code without any variables:

```
<html>
  <body>
    <script type="text/javascript">
      document.write( 0.872 * prompt( "How many Euros do you➥
 wish to convert", "" ) );
    </script>
  </body>
</html>
```

It's not immediately obvious that this code is converting euros to dollars, because there's nothing to tell us that 0.872 is the exchange rate. The code works fine though; if you try it out with the number 10, you should get the following result:

```
8.72
```

We are using the `prompt()` method of the `window` object to get user feedback in this example (the `window` is optional in this instance, and to keep the code shorter you can omit it). This method has two parameters: one label displayed above an entry field and an initial value of the field. You'll learn more about `prompt()` and how to use it in Chapter 4. Supposing that we wanted to make the result a little more informative, like this:

```
10 Euros is 8.72 Dollars
```

Without variables, the only way to do it would be to ask users to enter the euros they want to convert twice, and that really wouldn't be user friendly. Using variables, though, we can store the data temporarily, and then call it up as many times as we need to:

```
<html>
  <body>
    <script type="text/javascript">
      // Declare a variable holding the conversion rate
      var euroToDollarRate = 0.872;
```

```
      // Declare a new variable and use it to store the
      // number of euros
      var eurosToConvert = prompt( "How many Euros do you wish➥
 to convert", "" );
      // Declare a variable to hold the result of the euros
      // multiplied by the conversion
      var dollars = eurosToConvert * euroToDollarRate;
      // Write the result to the page
      document.write( eurosToConvert + " euros is " + dollars +➥
 " dollars" );
    </script>
  </body>
</html>
```

We've used three variables: one to store the exchange rate from euros to dollars, another to store the number of euros that will be converted, and the final one to hold the result of the conversion into dollars. Then all we need to do is write out the result using both variables. Not only is this script more functional, it's also much easier to read.

Converting Different Types of Data

For the most part, the JavaScript interpreter can work out what data types we want to be used. In the following code, for example, the interpreter understands the numbers 1 and 2 to be of number data type and treats them accordingly:

```
<html>
  <body>
    <script type="text/javascript">
      var myCalc = 1 + 2;
      document.write( "The calculated number is " + myCalc );
    </script>
  </body>
</html>
```

This will be written to your page:

```
The calculated number is 3
```

However, if we rewrite the code to allow the user to enter his own number using the `prompt()` function, then we'll get a different calculation altogether:

```
<html>
  <body>
    <script type="text/javascript">
      var userEnteredNumber = prompt( "Please enter a number",➥
 "" );
      var myCalc = 1 + userEnteredNumber;
      var myResponse = "The number you entered + 1 = " + myCalc;
      document.write( myResponse );
    </script>
  </body>
</html>
```

If you enter `2` at the prompt, then you'll be told that

```
The number you entered + 1 =  12
```

Rather than add the two numbers together, the JavaScript interpreter has concatenated them. This is because the `prompt()` function actually returns the value entered by the user as a string data type, even though the string contains number characters. The concatenation happens in this line:

```
var myCalc = 1 + userEnteredNumber;
```

In effect, it's the same as if we'd written

```
var myCalc = 1 + "2";
```

If, however, we use the subtraction operator instead:

```
var myCalc = 1 - userEnteredNumber;
```

`userEnteredNumber` is subtracted from 1. The subtraction operator isn't applicable to string data, so JavaScript works out that we wanted the data to be treated as a number, converts the string to a number, and does the calculation. The same applies to the * and / operators. The `typeof()` operator returns the type of data that has been passed to it, so we can use that to see which data types the JavaScript interpreter is working with:

```
<html>
  <body>
    <script type="text/javascript">
      var userEnteredNumber = prompt( "Please enter a number",➥
 "" );
```

```
      document.write( typeof( userEnteredNumber ) );
    </script>
  </body>
</html>
```

This will write string into the page. The way to ensure that the interpreter is using the desired number data type is to **explicitly** declare that the data is a number. There are three functions you can use to do this:

- `Number()`: Tries to convert the value of the variable inside the parentheses into a number.
- `parseFloat()`: Tries to convert the value to a floating point. It parses the string character by character from left to right, until it encounters a character that can't be used in a number. It then stops at that point and evaluates this string as a number. If the first character can't be used in a number, the result is **NaN** (which stands for **Not a Number**).
- `parseInt()`: Converts the value to an integer by removing any fractional part without rounding the number up or down. Anything nonnumerical passed to the function will be discarded. If the first character is not +, -, or a digit, the result is NaN.

Let's see how these functions work in practice:

```
<html>
  <body>
    <script type="text/javascript">
      var userEnteredNumber = prompt( "Please enter a number",➥
 "" );
      document.write( typeof( userEnteredNumber ) );
      document.write( "<br />" );
      document.write( parseFloat( userEnteredNumber ) );
      document.write( "<br />" );
      document.write( parseInt( userEnteredNumber ) );
      userEnteredNumber = Number( userEnteredNumber )
      document.write( "<br />" );
      document.write( userEnteredNumber );
      document.write( "<br />" );
      document.write( typeof( userEnteredNumber ) );
    </script>
  </body>
</html>
```

Try entering the value `23.50`. You should get this output:

```
string
23.5
23
23.5
number
```

The data entered is read as a string in the first line. Then `parseFloat()` converts 23.50 from a string to a floating point number, and in the next line `parseInt()` strips out the fractional part (without rounding up or down). The variable is then converted to a number using the `Number()` function, stored in the `userEnteredNumber` variable itself (overwriting the string held there), and on the final line we see that `userEnteredNumber`'s data type is indeed number.

Try entering `23.50abc` at the user prompt:

```
string
23.5
23
NaN
number
```

The results are similar, but this time `Number()` has returned NaN. The `parseFloat()` and `parseInt()` functions still return a number because they work from left to right converting as much of the string to a number as they can, and then stop when they hit a nonnumeric value. The `Number()` function will reject any string that contains nonnumerical characters (digits, a valid decimal place, and + and - signs are allowed but nothing else).

If you try entering `abc`, you'll just get

```
string
NaN
NaN
NaN
number
```

None of the functions can find a valid number, and so they all return NaN, which we can see is a number data type, but not a valid number. This is a good way of checking user input for validity, and we'll be using it to do exactly that later on.

So let's get back to the problem we started with: using `prompt()` to retrieve a number. All we need to do is tell the interpreter the data entered by the user should be converted to a number data type, using one of the functions discussed with the `prompt()` function:

```
<html>
  <body>
    <script type="text/javascript">
      var userEnteredNumber = Number( prompt( "Please enter➥
 a number", "" ) );
      var myCalc = 1 + userEnteredNumber;
      var myResponse = "The number you entered + 1 = " +➥
 myCalc;
      document.write( myResponse );
    </script>
  </body>
</html>
```

This will not throw any error, but it does not help the visitor much, as the meaning of NaN is not common knowledge. Later on we will deal with conditions, and you'll see how you could prevent an output that does not make much sense to the non-JavaScript-savvy user.

And that's all you need to know about primitive data types and variables for now. Primitive data types, as you have seen, simply hold a value. However, JavaScript can also deal with complex data, and it does this using **composite** data types.

The Composite Data Types: Array and Object

Composite data types are different from simple data types, as they can hold more than one value. There are two composite data types:

- **Object**: Contains a reference to any object, including the objects that the browser makes available
- **Array**: Contains one or more of any other data types

We'll look at the object data type first. As you might recall from the discussion in Chapter 1, objects model real-world entities. These objects can hold data and provide us with properties and methods.

Objects JavaScript Supplies You with: `String`, `Date`, and `Math`

These three objects do three different things:

- `String` **object**: Stores a string, and provides properties and methods for working with strings
- `Date` **object**: Stores a date, and provides methods for working with it
- `Math` **object**: Doesn't store data, but provides properties and methods for manipulating mathematical data

Let's start with the `String` object.

The `String` Object

Earlier we created string primitives by giving them some characters to hold, like this:

```
var myPrimitiveString = "ABC123";
```

A `String` **object** does things slightly differently, not only allowing us to store characters, but also providing a way to manipulate and change those characters. You can create `String` objects explicitly or implicitly.

Creating a `String` Object

Let's work with the implicit method first: we'll begin declaring a new variable and assign it a new string primitive to initialize it. Try that now using `typeof()` to make sure that the data in the variable `myStringPrimitive` is a string primitive:

```
<html>
  <body>
    <script type="text/javascript">
      var myStringPrimitive= "abc";
      document.write( typeof( myStringPrimitive ) );
    </script>
  </body>
</html>
```

We can still use the `String` object's methods on it though. JavaScript will simply convert the string primitive to a temporary `String` object, use the method on it, and then change the data type back to string. We can try that out using the `length` **property** of the `String` object:

```
<html>
  <body>
    <script type="text/javascript">
      var myStringPrimitive= "abc";
      document.write( typeof( myStringPrimitive ) );
      document.write( "<br>" );
      document.write( myStringPrimitive.length );
      document.write( "<br>" );
      document.write( typeof( myStringPrimitive ) );
    </script>
  </body>
</html>
```

This is what you should see in the browser window:

```
string
3
string
```

So myStringPrimitive is still holding a string primitive after the temporary conversion. We can also create String objects explicitly, using the new keyword together with the String() **constructor**:

```
<html>
  <body>
    <script type="text/javascript">
      var myStringObject = new String( "abc" );
      document.write( typeof( myStringObject ) );
      document.write( "<br />" );
      document.write( myStringObject.length );
      document.write( "<br />" );
      document.write( typeof( myStringObject ) );
    </script>
  </body>
</html>
```

Loading this page displays the following:

```
object
3
object
```

The only difference between this script and the previous one is in the first line where we create the new object and supply some characters for the String object to store:

```
var myStringObject = new String( "abc" );
```

The result of checking the length property is the same whether we create the String object implicitly or explicitly. The only real difference between creating String objects explicitly or implicitly is that creating them explicitly is marginally more efficient if you're going to be using the same String object again and again. Explicitly creating String objects also helps prevent the JavaScript interpreter getting confused between numbers and strings, as it can do.

Using the String Object's Methods

The String object has a lot of methods, so we'll limit our discussion to two of them here, the indexOf() and substring() methods.

JavaScript strings, as you've seen, are made up of characters. Each of these characters is given an index. The index is zero-based, so the first character's position has the index 0, the second 1, and so on. The method indexOf() finds and returns the position in the index at which a substring begins (and the lastIndexOf() method returns the position at which the substring ends). For example, if we want our user to enter an e-mail address, we could check that they'd included the @ symbol in their entry. (While this wouldn't ensure that the address is valid, it would at least go some way in that direction. We'll be working with much more complex data checking later on in the book.)

Let's do that next, using the `prompt()` method to obtain the user's e-mail address and then check the input for the `@` symbol, returning the index of the symbol using `indexOf()`:

```
<html>
    <body>
    <script type="text/javascript">
      var userEmail= prompt( "Please enter your email➥
 address ", "" );
      document.write( userEmail.indexOf( "@" ) );
    </script>
  </body>
</html>
```

If the `@` is not found, `-1` is written to the page. As long as the character is there in the string somewhere, its position in the index, in other words something greater than –1, will be returned.

The `substring()` method carves one string from another string, taking the indexes of the start and end position of the substring as parameters. We can return everything from the first index to the end of the string by leaving off the second parameter.

So to extract all the characters from the third character (at index 2) to the sixth character (index 5), we'd write

```
<html>
  <body>
    <script type="text/javascript">
      var myOldString = "Hello World";
      var myNewString = myOldString.substring( 2, 5 );
      document.write( myNewString );
    </script>
  </body>
</html>
```

You should see `llo` written out to the browser. Note that the `substring()` method copies the substring that it returns, and it doesn't alter the original string.

The `substring()` method really comes into its own when you're working with unknown values. Here's another example that uses both the `indexOf()` and `substring()` methods:

```
<html>
  <body>
    <script type="text/javascript">
      var characterName = "my name is Simpson,  Homer";
      var firstNameIndex = characterName.indexOf( "Simpson,➥
  " ) + 9;
      var firstName = characterName.substring( firstNameIndex );
      document.write( firstName );
    </script>
  </body>
</html>
```

We're extracting `Homer` from the string in the variable `characterName`, using `indexOf()` to find the start of the last name, and adding `9` to it to get the index of the start of the first name (as "`Simpson, `" is 9 characters long), and storing it in `firstNameIndex`. This is used by the `substring()` method to extract everything from the start of the first name—we haven't specified the final index, so the rest of the characters in the string will be returned.

Now let's look at the `Date` object. This allows us to store dates and provides some useful date/time-related functionality.

The `Date` Object

JavaScript doesn't have a primitive date data type, so we can only create `Date` objects explicitly. We create new `Date` objects the same way as we create `String` objects, using the `new` keyword together with the `Date()` constructor. This line creates a `Date` object containing the current date and time:

```
var todaysDate = new Date();
```

To create a `Date` object that stores a specific date or time, we simply put the date, or date and time, inside the parentheses:

```
var newMillennium = new Date( "1 Jan 2000 10:24:00" );
```

Different countries describe dates in a different order. For example, in the US dates are specified in *MM/DD/YY*, while in Europe they are *DD/MM/YY*, and in China they are *YY/MM/DD*. If you specify the month using the abbreviated name, then you can use any order:

```
var someDate = new Date( "10 Jan 2002" );
var someDate = new Date( "Jan 10 2002" );
var someDate = new Date( "2002 10 Jan" );
```

In fact, the `Date` object can take a number of parameters:

```
var someDate = new Date( aYear,  aMonth,  aDate,➥
anHour,  aMinute,  aSecond, aMillisecond )
```

To use these parameters, you first need to specify year and month, and then use the parameters you want—although you do have to run through them in order and can't select among them. For example, you can specify year, month, date, and hour:

```
var someDate = new Date( 2003,  9,  22,  17 );
```

You can't specify year, month, and then hours though:

```
var someDate = new Date( 2003,  9,  ,  17 );
```

■ **Note** Although we usually think of month 9 as September, JavaScript starts counting months from 0 (January), and so September is represented as month 8.

Using the Date Object

The Date object has a lot of methods that you can use to get or set a date or time. You can use local time (the time on your computer in your time zone) or UTC (Coordinated Universal Time, once called Greenwich Mean Time). While this can be very useful, you need to be aware when you're working with Date that many people don't set their time zone correctly.

Let's look at an example that demonstrates some of the methods:

```
<html>
  <body>
    <script type="text/javascript">
      // Create a new date object
      var someDate = new Date( "31 Jan 2003 11:59" );
      // Retrieve the first four values using the
      // appropriate get methods
      document.write( "Minutes = " + someDate.getMinutes() + "<br>" );
      document.write( "Year = " + someDate.getFullYear() + "<br>" );
      document.write( "Month = " + someDate.getMonth() + "<br>" );
      document.write( "Date = " + someDate.getDate() + "<br>" );
      // Set the minutes to 34
      someDate.setMinutes( 34 );
      document.write( "Minutes = " + someDate.getMinutes() + "<br>" );
      // Reset the date
      someDate.setDate( 32 );
      document.write( "Date = " + someDate.getDate() + "<br>" );
      document.write( "Month = " + someDate.getMonth() + "<br>" );
    </script>
  </body>
</html>
```

Here's what you should get:

```
Minutes = 59
Year = 2003
Month = 0
Date = 31
Minutes = 34
Date = 1
Month = 1
```

This line of code might look a bit counterintuitive at first:

```
someDate.setDate( 32 );
```

JavaScript knows that there aren't 32 days in January, so instead of trying to set the date to the January 32, the interpreter counts 32 days beginning with January 1, which gives us February 1.

This can be a handy feature if you need to add days onto a date. Usually we'd have to take into account the number of days in the different months, and whether it's a leap year, if we wanted to add a number of days to a date, but it's much easier to use JavaScript's understanding of dates instead:

```
<html>
  <body>
    <script type="text/javascript">
      // Ask the user to enter a date string
      var originalDate = prompt( Enter a date (Day, Name of➥
      the Month, Year), "31 Dec 2003" );
      // Overwrite the originalDate variable with a new Date
      // object
      var originalDate = new Date( originalDate );
      // Ask the user to enter the number of days to be
      // added,  and convert to number
      var addDays = Number( prompt( "Enter number of days➥
 to be added", "1" ) )
      // Set a new value for originalDate of originalDate
      // plus the days to be added
      originalDate.setDate( originalDate.getDate( ) + addDays )
      // Write out the date held by the originalDate
      // object using the toString( ) method
      document.write( originalDate.toString( ) )
    </script>
  </body>
</html>
```

If you enter `31 Dec 2003` when prompted, and `1` for the number of days to be added, then the answer you'll get is `Thu Jan 1 00:00:00 UTC 2004`.

■ **Note** Notice that we're using the `Number()` method of the `Math` object on the third line of the script. The program will still run if we don't, but the result won't be the same. If you don't want to use the method, there is a trick to convert different data types: if you subtract `0` from a string that could be converted to a number using `parseInt()`, `parseFloat()`, or `Number()`, then you convert it to a number, and if you add an empty string, `''`, to a number, you convert it to a string, something you normally do with `toString()`.

On the fourth line, we set the date to the current day of the month, the value returned by `originalDate.getDate()` plus the number of days to be added; then comes the calculation, and the final line outputs the date contained in the `Date` object as a string using the `toString()`

method. If you're using IE5.5+ or Gecko-based browsers (Mozilla, Netscape >6), `toDateString()` produces a nicely formatted string using the date alone. You can use the same methods for `get` and `set` if you're working with UTC time—all you need to do is add UTC to the method name. So `getHours()` becomes `getUTCHours()`, `setMonth()` becomes `setUTCMonth()`, and so on. You can also use the `getTimezoneOffset()` method to return the difference, in hours, between the computer's local time and UTC time. (You'll have to rely on users having set their time zones correctly, and be aware of the differences in daylight saving time between different countries.)

■ **Note** For crucial date manipulation, JavaScript might not be the correct technology, as you cannot trust the client computer to be properly set up. You could, however, populate the initial date of your JavaScript via a server-side language and go from there.

The `Math` Object

The `Math` object provides us with lots of mathematical functionality, like finding the square of a number or producing a random number. The `Math` object is different from the `Date` and `String` objects in two ways:

- You can't create a `Math` object explicitly, you just go ahead and use it.
- The `Math` object doesn't store data, unlike the `String` and `Date` object.

You call the methods of the `Math` object using the format:

```
Math.methodOfMathObject( aNumber ):
alert( "The value of pi is " + Math.PI );
```

We'll look at a few of the commonly used methods next (you can find a complete reference by running a search at `http://www.mozilla.org/docs/web-developer/`). We'll look at the methods for rounding numbers and generating random numbers here.

Rounding Numbers

You saw earlier that the `parseInt()` function will make a fractional number whole by removing everything after the decimal point (so 24.999 becomes 24). Pretty often you'll want more mathematically accurate calculations, if you're working with financial calculations, for example, and for these you can use one of the `Math` object's three rounding functions: `round()`, `ceil()`, and `floor()`. This is how they work:

- `round()`: Rounds a number up when the decimal is . 5 or greater
- `ceil()` **(as in ceiling)**: Always rounds up, so 23.75 becomes 24, as does 23.25
- `floor()`: Always rounds down, so 23.75 becomes 23, as does 23.25

Here they are at work in a simple example:

```
<html>
  <body>
    <script type="text/javascript">
      var numberToRound = prompt( "Please enter a number", "" )
      document.write( "round( ) = " + Math.round( numberToRound ) );
      document.write( "<br>" );
      document.write( "floor( ) = " + Math.floor( numberToRound ) );
      document.write( "<br>" );
      document.write( "ceil( ) = " + Math.ceil( numberToRound ) );
    </script>
  </body>
</html>
```

Even though we used `prompt()` to obtain a value from the user, which as we saw earlier returns a string, the number returned is still treated as a number. This is because the rounding methods do the conversion for us just so long as the string contains something that can be converted to a number.

If we enter `23.75`, we get the following result:

```
round() = 24
floor() = 23
ceil() = 24
```

If we enter `-23.75`, we get

```
round() = -24
floor() = -24
ceil() = -23
```

Generating a Random Number

You can generate a fractional random number that is 0 or greater but smaller than 1 using the `Math` object's `random()` method. Usually you'll need to multiply the number, and then use one of the rounding methods in order to make it useful.

For example, in order to mimic a die throw, we'd need to generate a random number between 1 and 6. We could create this by multiplying the random fraction by 5, to give a fractional number between 0 and 5, and then round the number up or down to a whole number using the `round()` method. (We couldn't just multiply by 6 and then round up every time using `ceil()`, because that would give us the occasional 0.) Then we'd have a whole number between

0 and 5, so by adding 1, we can get a number between 1 and 6. This approach won't give you a perfectly balanced die, but it's good enough for most purposes. Here's the code:

```
<html>
  <body>
    <script type="text/javascript">
      var diceThrow = Math.round( Math.random( ) * 5 ) + 1;
      document.write( "You threw a " + diceThrow );
    </script>
  </body>
</html>
```

Arrays

JavaScript allows us to store and access related data using an **array**. An array is a bit like a row of boxes (**elements**), each box containing a single item of data. An array can work with any of the data types that JavaScript supports. So, for example, you could use an array to work with a list of items that the users will select from, or for a set of graph coordinates, or to reference a group of images.

`Array` objects, like `String` and `Date` objects, are created using the `new` keyword together with the constructor. We can initialize an `Array` object when we create it:

```
var preInitArray = new Array( "First item", "Second item",➥
  "Third Item" );
```

Or set it to hold a certain number of items:

```
var preDeterminedSizeArray = new Array( 3 );
```

Or just create an empty array:

```
var anArray = new Array();
```

You can add new items to an array by assigning values to the elements:

```
anArray[0] = "anItem"
anArray[1] = "anotherItem"
anArray[2] = "andAnother"
```

■ **Tip** You do not have to use the `array()` constructor; instead it is perfectly valid to use a shortcut notation:

```
var myArray = [1, 2, 3];
var yourArray = ["red", "blue", "green"]
```

Once we've populated an array, we can access its elements through their indexes or positions (which, once again, are zero-based) using square brackets:

```
<html>
  <body>
    <script type="text/javascript">
      var preInitArray = new Array( "First Item",➥
  "Second Item", "Third Item" );
      document.write( preInitArray[0] + "<br>" );
      document.write( preInitArray[1] + "<br>" );
      document.write( preInitArray[2] + "<br>" );
    </script>
  </body>
</html>
```

Using index numbers to store items is useful if you want to loop through the array—we'll be looking at loops next.

You can use keywords to access the array elements instead of a numerical index, like this:

```
<html>
  <body>
    <script type="text/javascript">
      // Creating an array object and setting index
      // position 0 to equal the string Fruit
      var anArray = new Array( );
      anArray[0] = "Fruit";
      // Setting the index using the keyword
      // 'CostOfApple' as the index.
      anArray["CostOfApple"] = 0.75;
      document.write( anArray[0] + "<br>" );
      document.write( anArray["CostOfApple"] );
    </script>
  </body>
</html>
```

Keywords are good for situations where you can give the data useful labels, or if you're storing entries that are only meaningful in context, like a list of graph coordinates. You can't, however, access entries using an index number if they have been set using keywords (as you can in some other languages, like PHP). We can also use variables for the index. We can rewrite the previous example using variables (one holding a string and the other a number), instead of literal values:

```
<html>
  <body>
    <script type="text/javascript">
      var anArray = new Array( );
      var itemIndex = 0;
      var itemKeyword = "CostOfApple";
```

```
      anArray[itemIndex] = "Fruit";
      anArray[itemKeyword] = 0.75;
      document.write( anArray[itemIndex] + "<br>" );
      document.write( anArray[itemKeyword] );
    </script>
  </body>
</html>
```

Let's put what we've discussed about arrays and the `Math` object into an example. We'll write a script that randomly selects a banner to display at the top of the page.

We'll use an `Array` object to hold some image source names, like this:

```
var bannerImages = new Array();
bannerImages[0] = "Banner1.jpg";
bannerImages[1] = "Banner2.jpg";
bannerImages[2] = "Banner3.jpg";
bannerImages[3] = "Banner4.jpg";
bannerImages[4] = "Banner5.jpg";
bannerImages[5] = "Banner6.jpg";
bannerImages[6] = "Banner7.jpg";
```

Then we need six images with corresponding names to sit in the same folder as the HTML page. You can use your own or download mine from `http://www.beginningjavascript.com`.

Next we'll initialize a new variable, `randomImageIndex`, and use it to generate a random number. We'll use the same method that we used to generate a random die throw earlier, but without adding 1 to the result because we need a random number from 0 to 6:

```
var randomImageIndex = Math.round( Math.random( ) * 6 );
```

Then we'll use `document.write()` to write the randomly selected image into the page. Here's the complete script:

```
<html>
  <body>
    <script type="text/javascript">
      var bannerImages = new Array( );
      bannerImages[0] = "Banner1.jpg";
      bannerImages[1] = "Banner2.jpg";
      bannerImages[2] = "Banner3.jpg";
      bannerImages[3] = "Banner4.jpg";
      bannerImages[4] = "Banner5.jpg";
      bannerImages[5] = "Banner6.jpg";
      bannerImages[6] = "Banner7.jpg";
      var randomImageIndex = Math.round( Math.random( ) * 6 );
      document.write( "<img alt=\"\" src=\"" + ➥
bannerImages[randomImageIndex] + "\">" );
    </script>
  </body>
</html>
```

And that's all there is to it. Having the banner change will make it more noticeable to visitors than if you displayed the same banner every time they came to the page—and, of course, it gives the impression that the site is being updated frequently.

The Array Object's Methods and Properties

One of the most commonly used properties of the Array object is the length property, which returns the index one count higher than the index of the last array item in the array. If, for example, you're working with an array with elements with indexes of 0, 1, 2, 3, the length will be 4—which is useful to know if you want to add another element.

The Array object provides a number of methods for manipulating arrays, including methods for cutting a number of items from an array, or joining two arrays together. We'll look at the methods for concatenating, slicing, and sorting next.

Cutting a Slice of an Array

The slice() method is to an Array object what the substring() method is to a String object. You simply tell the method which elements you want to be sliced. This would be useful, for example, if you wanted to slice information being passed using a URL.

The slice() method takes two parameters: the index of the first element of the slice, which will be included in the slice, and the index of the final element, which won't be. To access the second, third, and fourth values from an array holding five values in all, we use the indexes 1 and 4:

```
<html>
  <body>
    <script type="text/javascript">
      // Create and initialize the array
      var fullArray = new Array( "One", "Two", "Three",➥
 "Four", "Five" );
      // Slice from element 1 to element 4 and store
      // in new variable sliceOfArray
      var sliceOfArray = fullArray.slice( 1, 4 );
      // Write out new ( zero-based ) array of 3 elements
      document.write( sliceOfArray[0] + "<br>" );
      document.write( sliceOfArray[1] + "<br>" );
      document.write( sliceOfArray[2] + "<br>" );
    </script>
  </body>
</html>
```

The new array stores the numbers in a new zero-based array, so slicing indexes 0, 1, and 2 gives us the following:

```
Two
Three
Four
```

The original array is unaffected, but you could overwrite the `Array` object in the variable by setting it to the result of the `slice()` method if you needed to:

```
fullArray = fullArray.slice( 1, 4 );
```

Joining Two Arrays

The `Array` object's `concat()` method allows us to concatenate arrays. We can add two or more arrays using this method, each new array starting where the previous one ends. Here we're joining three arrays: `arrayOne`, `arrayTwo`, and `arrayThree`:

```
<html>
  <body>
    <script type="text/javascript">
      var arrayOne = new Array( "One", "Two", "Three",➥
 "Four", "Five" );
      var arrayTwo = new Array( "ABC", "DEF", "GHI" );
      var arrayThree = new Array( "John", "Paul", "George",➥
 "Ringo" );
      var joinedArray = arrayOne.concat( arrayTwo, arrayThree );
      document.write( "joinedArray has " + joinedArray.length + ➥
" elements<br>" );
      document.write( joinedArray[0] + "<br>" )
      document.write( joinedArray[11] + "<br>" )
    </script>
  </body>
</html>
```

The new array, `joinedArray`, has 12 items. The items in this array are the same as they were in each of the previous arrays; they've simply been concatenated together. The original arrays remain untouched.

Converting an Array to a String and Back

Having data in an array is dead handy when you want to loop through it or select certain elements. However, when you need to send the data somewhere else, it might be a good idea to convert that data to a string. You could do that by looping through the array and adding each element value to a string. However, there is no need for that, as the `Array` object has a method called `join()` that does that for you. The method takes a string as a parameter. This string will be added in between each element.

```
<script type="text/javascript">
  var arrayThree = new Array( "John", "Paul", "George",➥
 "Ringo" );
  var lineUp=arrayThree.join( ',  ' );
  alert( lineUp );
</script>
```

The resulting string, `lineUp`, has the value `"John, Paul, George, Ringo"`. The opposite of `join()` is `split()`, which is a method that converts a string to an array.

```
<script type="text/javascript">
  var lineUp="John,  Paul,  George,  Ringo";
  var members=lineUp.split( ',  ' );
  alert( members.length );
</script>
```

Sorting an Array

The `sort()` method allows us to sort the items in an array into alphabetical or numerical order:

```
<html>
  <body>
    <script type="text/javascript">
      var arrayToSort = new Array( "Cabbage", "Lemon",➥
 "Apple", "Pear", "Banana" );
      var sortedArray = arrayToSort.sort( );
      document.write( sortedArray[0] + "<br>" );
      document.write( sortedArray[1] + "<br>" );
      document.write( sortedArray[2] + "<br>" );
      document.write( sortedArray[3] + "<br>" );
      document.write( sortedArray[4] + "<br>" );
    </script>
  </body>
</html>
```

The items are arranged like this:

```
Apple
Banana
Cabbage
Lemon
Pear
```

If, however, you lower the case of one of the letters—the *A* of *Apple*, for example—then you'll end up with a very different result. The sorting is strictly mathematical—by the number of the character in the ASCII set, not like a human being would sort the words.

If you wanted to change the order the sorted elements are displayed, you can use the `reverse()` method to display the last in the alphabet as the first element:

```
<script type="text/javascript">
  var arrayToSort = new Array( "Cabbage", "Lemon", ➥
"Apple", "Pear", "Banana" );
  var sortedArray = arrayToSort.sort( );
  var reverseArray = sortedArray.reverse( );
  document.write( reverseArray[0] + "<br />" );
  document.write( reverseArray[1] + "<br />" );
  document.write( reverseArray[2] + "<br />" );
  document.write( reverseArray[3] + "<br />" );
  document.write( reverseArray[4] + "<br />" );
</script>
```

The resulting list is now in reverse order:

```
Pear
Lemon
Cabbage
Banana
Apple
```

Making Decisions in JavaScript

Decision making is what gives programs their apparent intelligence. You can't write a good program without it, whether you're creating a game, checking a password, giving the user a set of choices based on previous decisions they have made, or something else.

Decisions are based on conditional statements, which are simply statements that evaluate to true or false. This is where the primitive Boolean data type comes in useful. Loops are the other essential tool of decision making, enabling you to loop through user input or an array, for example, and make decisions accordingly.

The Logical and Comparison Operators

There are two main groups of operators we'll look at:

- **Data comparison operators**: Compare operands and return Boolean values.
- **Logical operators**: Test for more than one condition.

We'll start with the comparison operators.

Comparing Data

Table 2-3 lists some of the more commonly used comparison operators.

Table 2-3. *Comparisons in JavaScript*

Operator	Description	Example
==	Checks whether the left and right operands are equal	123 == 234 returns false. 123 == 123 returns true.
!=	Checks whether the left operand is not equal to the right side	123 != 123 returns false. 123 != 234 returns true.
>	Checks whether the left operand is greater than the right	123 > 234 returns false. 234 > 123 returns true.
>=	Checks whether the left operand is greater than or equal to the right	123 >= 234 returns false. 123 >= 123 returns true.
<	Checks whether the left operand is less than the right	234 < 123 returns false. 123 < 234 returns true.
<=	Checks whether the left operand is less than, or equal to, the right	234 <= 123 returns false. 234 <= 234 returns true.

■ **Caution** Beware the == equality operator: it's all too easy to create errors in a script by using the assignment operator, =, by mistake.

These operators all work with string type data as well as numerical data, and are case sensitive:

```
<html>
  <body>
    <script type="text/javascript">
      document.write( "Apple" == "Apple" )
      document.write( "<br />" );
      document.write( "Apple" < "Banana" )
      document.write( "<br />" );
      document.write( "apple" < "Banana" )
    </script>
  </body>
</html>
```

This is what you should get back:

```
true
true
false
```

When evaluating an expression comparing strings, the JavaScript interpreter compares the ASCII codes for each character in turn of both strings—the first character of each string, then the second character, and so on. Uppercase *A* is represented in ASCII by the number 65, *B* by 66, *C* by 67, and so on. To evaluate the expression `"Apple" < "Banana"`, the JavaScript interpreter tests the comparison by substituting the ASCII code for the first character in each string: `65 < 66`, so *A* sorts first, and the comparison is true. When testing the expression `"apple" < "Banana"`, the JavaScript interpreter does the same thing; however, the lowercase letter *a* has the ASCII code 97, so the expression `"a" < "B"` reduces to `97 < 66`, which is false. You can do alphabetical comparisons using <, <=, >, >= operators. If you need to ensure that all the letters are of the same case, you can use the `String` object's `toUpperCase()` and `toLowerCase()` methods. Comparison operators, just like the numerical operators, can be used with variables. If we wanted to compare `apple` and `Banana` alphabetically, we'd do this:

```
<html>
  <body>
    <script type="text/javascript">
      var string1 = "apple";
      var string2 = "Banana";
      string1 = string1.toLowerCase( );
      string2 = string2.toLowerCase( );
      document.write( string1 < string2 )
    </script>
  </body>
</html>
```

There is something else you need to be aware of when you're comparing `String` objects using the equality operator, though. Try this:

```
<html>
  <body>
    <script type="text/javascript">
      var string1 = new String( "Apple" );
      var string2 = new String( "Apple" );
      document.write( string1 == string2 )
    </script>
  </body>
</html>
```

You'll get `false` returned. In fact, what we've done here is compare two `String` *objects* rather than the *characters* of two string primitives and, as the returned `false` indicates, two `String` objects can't be the same object even if they do hold the same characters.

If you do need to compare the strings held by two objects, then you can use the `valueOf()` method to perform a comparison of the data values:

```
<html>
  <body>
    <script type="text/javascript">
      var string1 = new String( "Apple" );
      var string2 = new String( "Apple" );
      document.write( string1.valueOf() == string2.valueOf() );
    </script>
  </body>
</html>
```

Logical Operators

Sometimes you'll need to combine comparisons into one condition group. You might want to check that the information users have given makes sense, or restrict the selections they can make according to their earlier answers. You can do this using the logical operators shown in Table 2-4.

Table 2-4. *Logical Operators in JavaScript*

Symbol	Operator	Description	Example

&&	And	Both conditions must be true.	`123 == 234 && 123 < 20 (false)`
			`123 == 234 && 123 == 123 (false)`
			`123 == 123 && 234 < 900 (true)`

||	Or	Either or both of the conditions must be true.	`123 == 234 || 123 < 20 (false)`
			`123 == 234 || 123 == 123 (true)`
			`123 == 123 || 234 < 900 (true)`

!	Not	Reverses the logic.	`!(123 == 234) (true)` `!(123 == 123) (false)`

Once we've evaluated the data, we need to be able to make decisions according to the outcome. This is where conditional statements and loops come in useful. You'll find that the operators that we've looked at in this chapter are most often used in the context of a conditional statement or loop.

Conditional Statements

The `if...else` structure is used to test conditions and looks like this:

```
if ( condition ) {
// Execute code in here if condition is true
} else {
// Execute code in here if condition is false
}
// After if/else code execution resumes here
```

If the condition being tested is true, the code within the curly braces following the `if` will be executed, but won't if it isn't. You can also create a block of code to execute should the condition set out in the `if` *not* be met, by using a final `else` statement.

Let's improve on the currency exchange converter we built earlier on in the chapter and create a loop to deal with nonnumeric input from the user:

```
<html>
  <body>
    <script type="text/javascript">
      var euroToDollarRate = 0.872;
      // Try to convert the input into a number
      var eurosToConvert = Number( prompt( "How many Euros➥
 do you wish to convert", "" ) );
      // If the user hasn't entered a number,  then NaN
      // will be returned
      if ( isNaN( eurosToConvert ) ) {
        // Ask the user to enter a value in numerals
        document.write( "Please enter the number in numerals" );
        // If NaN is not returned,  then we can use the input
      } else {
        // and do the conversion as before
        var dollars = eurosToConvert * euroToDollarRate;
        document.write( eurosToConvert + " euros is " + ➥
dollars + " dollars" );
      }
    </script>
  </body>
</html>
```

The `if` statement is using the `isNaN()`function, which will return `true` if the value in variable `eurosToConvert` is not a number.

■ **Note** Remember to keep error messages as polite and helpful as possible. Good error messages that inform users clearly what is expected of them make using applications much more painless.

We can create more complex conditions by using logical operators and nesting `if` statements:

```
<html>
  <body>
    <script type="text/javascript">
      // Ask the user for a number and try to convert the
      // input into a number
      var userNumber = Number( prompt( "Enter a number between➥
 1 and 10", "" ) );
        // If the value of userNumber is NaN,  ask the user
        // to try again
      if ( isNaN( userNumber ) ) {
        document.write( "Please ensure a valid number is ➥
entered" );
        // If the value is a number but over 10,  ask the
        //user to try again
      } else {
        if ( userNumber > 10 || userNumber < 1 ) {
          document.write( "The number you entered is not➥
 between 1 and 10" );
        // Otherwise the number is between 1 and 10 so
        // write to the page
        } else {
          document.write( "The number you entered was " + userNumber );
        }
      }
    </script>
  </body>
</html>
```

We know that the number is fine just so long as it is a numeric value and is under 10.

■ **Note** Observe the layout of the code. We have indented the `if` and `else` statements and blocks so that it's easy to read and to see where code blocks start and stop. It's essential to make your code as legible as possible.

Try reading this code without the indenting or spacing:

```
<html>
<body>
<script type="text/javascript">
// Ask for a number using the prompt() function and try to make it a number
var userNumber = Number(prompt("Enter a number between 1 and 10",""));
// If the value of userNumber is NaN, ask the user to try again
if (isNaN(userNumber)){
document.write("Please ensure a valid number is entered");
}
// If the value is a number but over 10, ask the user to try again
else {
if (userNumber > 10 || userNumber < 1) {
document.write("The number you entered is not between 1 and 10");
}
// Otherwise the number is between 1 and 10, so write to the screen
else{
document.write("The number you entered was " + userNumber);
}
}
</script>
</body>
</html>
```

It's not impossible to read, but even in this short script, it's harder to decipher which code blocks belong to the `if` and `else` statements. In longer pieces of code, inconsistent indenting or illogical indenting makes code very difficult to read, which in turn leaves you with more bugs to fix and makes your job unnecessarily harder.

You can also use `else if` statements, where the `else` statement starts with another `if` statement, like this:

```
<html>
  <body>
    <script type="text/javascript">
      var userNumber = Number( prompt( "Enter a number between➥
 1 and 10", "" ) );
      if ( isNaN( userNumber ) ){
        document.write( "Please ensure a valid number is➥
 entered" );
      } else if ( userNumber > 10 || userNumber < 1 ) {
        document.write( "The number you entered is not➥
 between 1 and 10" );
      } else {
```

```
        document.write( "The number you entered was " + ➥
userNumber );
      }
    </script>
  </body>
</html>
```

The code does the same thing as the earlier piece, but uses an `else if` statement instead of a nested `if`, and is two lines shorter.

Breaking Out of a Branch or Loop

One more thing before we move on: you can break a conditional statement or loop using the `break` statement. This simply terminates the block of code running and drops the processing through to the next statement. We'll be using this in the next example.

You can have as many `if`, `else`, and `else if`s as you like, although they can make your code terribly complicated if you use too many. If there are a lot of possible conditions to check a value against in a piece of code, then the `switch` statement, which we'll look at next, can be helpful.

Testing Multiple Values: the `switch` Statement

The `switch` statement allows us to "switch" between sections of code based on the value of a variable or expression. This is the outline of a `switch` statement:

```
switch( expression ) {
  case someValue:
  // Code to execute if expression == someValue;
  break; // End execution
  case someOtherValue:
  // Code to execute if expression == someOtherValue;
  break; // End execution
  case yesAnotherValue:
  // Code to execute if expression == yetAnotherValue;
  break; // End execution
  default:
  // Code to execute if no values matched
}
```

JavaScript evaluates `switch (expression)`, and then compares it to each `case`. As soon as a match is found, the code starts executing at that point and continues through all the `case` statements until a `break` is found. It's often useful to include a default `case` that will execute if none of the `case` statements match. This is a helpful tool for picking up on errors, where, for example, we expect a match to occur but a bug prevents that from happening.

The values of the cases can be of any data type, numbers or strings, for example. We can have just one or as many cases as we need. Let's look at a simple example:

```
<html>
  <body>
    <script type="text/javascript">
      // Store user entered number between 1 and 4 in userNumber
      var userNumber = Number( prompt( "Enter a number between 1 and 4", "" ) );
      switch( userNumber ) {
        // If userNumber is 1,  write out and carry on
        // executing after case statement
        case 1:
          document.write( "Number 1" );
        break;
        case 2:
          document.write( "Number 2" );
        break;
        case 3:
          document.write( "Number 3" );
        break;
        case 4:
          document.write( "Number 4" );
        break;
        default:
          document.write( "Please enter a numeric value between➥
 1 and 4." );
        break;
      }
      // Code continues executing here
    </script>
  </body>
</html>
```

Try it out. You should just get the number that you've entered written out or the sentence "Please enter a numeric value between 1 and 4."

This example also illustrates the importance of the `break` statement. If we hadn't included `break` after each case, execution would have carried on within the block until the end of the `switch`. Try removing the `break`s and then enter `2`. Everything after the match will execute, giving you this output:

```
Number 2Number 3Number 4Please enter a numeric value between 1 and 4
```

You can use any valid expression inside the `switch` statement—a calculation, for example:

```
switch( userNumber * 100 + someOtherVariable )
```

You can also have one or more statements in between the `case` statements.

Repeating Things: Loops

In this section, we look at how we can repeat a block of code for as long as a set condition is true. For example, we might want to loop through each `input` element on an HTML form or through each item in an array.

Repeating a Set Number of Times: the `for` Loop

The `for` loop is designed to loop through a code block a number of times and looks like this:

```
for( initial-condition; loop-condition; alter-condition ) {
  //
  // Code to be repeatedly executed
  //
}
// After loop completes,  execution of code continues here
```

Like the conditional statement, the `for` keyword is followed by parentheses. This time, the parentheses contain three parts separated by a semicolon.

The first part initializes a variable that will serve as the counter to keep track of the number of loops made. The second part tests for a condition. The loop will keep running as long as this condition is true. The last part either increments or decrements the counter, created in the first part, after each loop (the fact that it is *after* is an important one in programming as you shall see).

For example, take a look at a loop that keeps running for as long as `loopCounter` is less than 10:

```
for( loopCounter = 1; loopCounter <= 10; loopCounter++ )
```

The loop keeps executing as long as the loop condition evaluates to true—for as long as `loopCounter` is less than or equal to 10. Once it hits 11, the looping stops and execution of the code continues at the next statement after the loop's closing parenthesis.

Let's look at an example that uses the `for` loop to run through an array. We'll use a `for` loop to run through an array called `theBeatles` using a variable called `loopCounter` to keep the loop running while the value of `loopCounter` is less than the length of the array:

```
<html>
  <body>
    <script type="text/javascript">
      var theBeatles = new Array( "John", "Paul", "George",➥
 "Ringo" );
```

```
      for ( var loopCounter = 0; loopCounter < ➥
theBeatles.length; loopCounter++ ) {
        document.write( theBeatles[loopCounter] + "<br>" );
      }
    </script>
  </body>
</html>
```

This example works because we are using a zero-based array in which the items have been added to the index in sequence. The loop wouldn't have run if we'd used keywords to store items in an array like this:

```
theBeatles["Drummer"] = "Ringo";
```

Earlier, when we discussed arrays, I stated that the `Array` object has a property that knows the length (how many elements). When looping through arrays, such as the previous example, we use the name of the array followed by a dot and length as the condition. This prevents the loop from counting beyond the length of the array, which would cause an "Out of Bounds" error.

JavaScript also supports the `for..in` loop (which has been around since NN2, although IE has supported it only since IE5). Instead of using a counter, the `for...in` loop runs though each item in the array using a variable to access the array. Let's create an array this way and see how it works:

```
<html>
  <body>
    <script type="text/javascript">
      // Initialize theBeatles array and store in a variable
      var theBeatles = new Array( );
      // Set the values using keys rather than numbers
      theBeatles["Drummer"] = "Ringo";
      theBeatles["SingerRhythmGuitar"] = "John";
      theBeatles["SingerBassGuitar"] = "Paul";
      theBeatles["SingerLeadGuitar"] = "George";
      var indexKey;
      // Write out each indexKey and the value for that
      // indexKey from the array
      for ( indexKey in theBeatles ) {
        document.write( "indexKey is " + indexKey + "<br>" );
        document.write( "item value is " + theBeatles[indexKey] + "<br><br>" );
      }
    </script>
  </body>
</html>
```

The results of the item key in `indexKey` at each iteration of the loop is written out alongside the value extracted from the array using that key in the same order as it occurs in the array:

```
indexKey is Drummer
item value is Ringo

indexKey is SingerRhythmGuitar
item value is John

indexKey is SingerBassGuitar
item value is Paul

indexKey is SingerLeadGuitar
item value is George
```

Repeating Actions According to a Decision: the `while` Loop

The loops we have been working with so far take the instruction to stop looping from inside the script itself. There are likely to be times when you'll want the user to determine when the loop should stop, or for the loop to stop when a certain user-led condition is met. The `while` and `do...while` loops are intended for just this sort of situation.

In its simplest form, a `while` loop looks like this:

```
while ( some condition true ) {
  // Loop code
}
```

The condition inside the curly braces can be anything you might use in an `if` statement. We could use some code like this to allow users to enter numbers and stop the entering process by typing the number 99.

```
<html>
  <body>
    <script type="text/javascript">
      var userNumbers = new Array( );
      var userInput = 0;
      var arrayIndex = 0;
      var message = '';
      var total = 0;
      // Loop for as long as the user doesn't input 99
      while ( userInput != 99 ) {
        userInput = prompt( "Enter a number,  or 99 to exit",➥
 "99" );
        userNumbers[arrayIndex] = userInput;
        arrayIndex++;
      }
```

```
      message += 'You entered the following:\n';
      for ( var i = 0; i < arrayIndex-1; i++ ) {
        message += userNumbers[i] + '\n';
        total += Number( userNumbers[i] );
      }
      message += 'Total: ' + total + '\n';
      alert( message );
    </script>
  </body>
</html>
```

Here the `while` loop's condition is that `userInput` is not equal to `99`, so the loop will continue so long as that condition is true. When the user enters `99` and the condition is tested, it will evaluate to false and the loop will end. Note the loop doesn't end as soon as the user enters `99`, but only when the condition is tested again at the start of another iteration of the loop.

There is one small but significant difference between the `while` loop and the `do...while`: the `while` loop tests the condition before the code is executed, and only executes the code block if the condition is true, while the `do...while` loop executes the code block before testing the condition, only doing another iteration if the condition is true. In short, the `do...while` loop is useful when you know you want the loop code to execute at least once before the condition is tested. We could write our previous example with a `do...while` loop like this:

```
<html>
  <body>
    <script type="text/javascript">
      var userNumbers = new Array( );
      var message = '';
      var total = 0;
      // Declare the userInput but don't initialize it
      var userInput;
      var arrayIndex = 0;
      do {
        userInput = prompt( "Enter a number,  or 99 to exit",➥
 "99" );
        userNumbers[arrayIndex] = userInput;
        arrayIndex++;
      } while ( userInput != 99 )
      message+='You entered the following:\n';
      for ( var i = 0; i < arrayIndex-1; i++ ) {
        message += userNumbers[i] + '\n';
        total += Number( userNumbers[i] );
      }
      message += 'Total: ' + total + '\n';
      alert( message );
    </script>
  </body>
</html>
```

We don't need to initialize `userInput` because the code inside the loop sets a value for it before testing it for the first time.

Continuing the Loop

As you've already seen, the `break` statement is great for breaking out of any kind of loop once a certain event has occurred. The `continue` keyword works like `break` in that it stops the execution of the loop. However, instead of dropping out of the loop, `continue` causes execution to resume with the next iteration.

Let's first alter the previous example so that if the user enters something other than a number, the value is not recorded and the loop finishes, using `break`:

```
<html>
  <body>
    <script type="text/javascript">
      var userNumbers = new Array( );
      var userInput;
      var arrayIndex = 0;
      do {
        userInput = Number( prompt( "Enter a number, or 99➥
 to exit", "99" ) );
        // Check that user input is a valid number,
        // and if not,  break with error msg
        if ( isNaN( userInput ) ) {
          document.write( "Invalid data entered: please➥
 enter a number between 0 and 99 in numerals" );
          break;
        }
        // If break has been activated,  code will continue from here
        userNumbers[arrayIndex] = userInput;
        arrayIndex++;
      } while ( userInput != 99 )
      // Next statement after loop
    </script>
  </body>
</html>
```

Now let's change it again, so that we don't break out of the loop but instead just ignore the user's input and keep looping, using the `continue` statement:

```
<html>
  <body>
    <script type="text/javascript">
      var userNumbers = new Array( );
      var userInput;
      var arrayIndex = 0;
      do {
        userInput = prompt( "Enter a number, or 99 to exit",➥
 "99" );
```

```
        if ( isNaN( userInput ) ) {
          document.write( "Invalid data entered: please ➥
enter a number between 0 and 99 in numerals " );
          continue;
        }
        userNumbers[arrayIndex] = userInput;
        arrayIndex++;
      } while ( userInput != 99 )
      // Next statement after loop
    </script>
  </body>
</html>
```

The `break` statement has been replaced with `continue`, so no more code will be executed in the loop, and the condition inside the `while` statement will be evaluated again. If the condition is true, another iteration of the loop will occur; otherwise the loop ends.

Use the following rules of thumb in deciding which looping structure to use:

- Use a `for` loop if you want to repeat an action a set number of times.
- Use a `while` loop when you want an action to be repeated until a condition is met.
- Use a `do...while` loop if you want to guarantee that the action will be performed at least once.

Summary

We've covered a lot of ground in this chapter: in fact, we've discussed most of the essentials of the JavaScript language.

You've learned how JavaScript handles data and seen that there are a number of data types: string, number, Boolean, and object, as well as some special types of data like NaN, null, and undefined. You saw that JavaScript supplies a number of operators that perform operations on data, such as numerical calculations or joining strings together.

We then looked at how JavaScript allows us to store values using variables. Variables last for the lifetime of the page, or the lifetime of the function if they are inside a user-created function and declared locally via the `var` keyword. We also looked at how to convert one type of data to another.

Next we worked with three JavaScript built-in objects: the `String`, `Date`, and `Math` objects. You saw that these provide useful functionality for manipulating strings, dates, and numbers. I also showed you the `Array` object, which allows a number of items of data to be stored in a single variable.

We finished the chapter by looking at decision making, which provides the logic or intelligence of a programming language. We used `if` and `switch` statements to make decisions, using conditions to test the validity of data and acting upon the findings. Loops also use conditions and allow us to repeat a block of code for a certain number of times or while a condition is true.