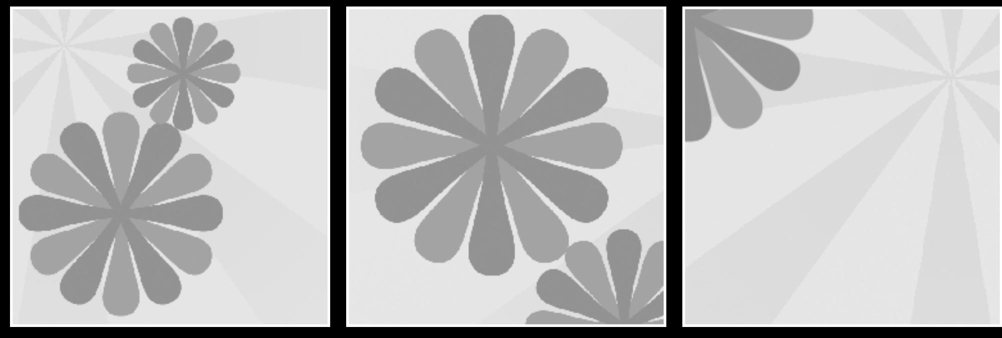
2 JAVASCRIPT SYNTAX



What this chapter covers:

- Statements
- Variables and arrays
- Operators
- Conditional statements and looping statements
- Functions and objects

This chapter is a brief refresher in JavaScript syntax, taking on the most important concepts.

What you'll need

You don't need any special software to write JavaScript. All you need is a plain text editor and a web browser.

Code written in JavaScript must be executed from a document written in (X)HTML. There are two ways of doing this. You can place the JavaScript between `<script>` tags within the `<head>` of the document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<script type="text/javascript">
JavaScript goes here...
</script>
</head>
<body>
Mark-up goes here...
</body>
</html>
```

A much better technique, however, is to place your JavaScript code into a separate file. Save this file with the file extension `.js`. You can then use the `src` attribute in a `<script>` tag to point to this file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<script type="text/javascript" src="file.js">
</script>
</head>
<body>
Mark-up goes here...
</body>
</html>
```

If you'd like to try the examples in this chapter, go ahead and create two files in a text editor. First, create a simple bare-bones HTML or XHTML file. You can call it something like `test.html`. Make sure that it contains a `<script>` tag in the `<head>` that has a `src` attribute with a value like `example.js`. That's the second file you can create in your text editor.

Your `test.html` file should look something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
 <head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Just a test</title>
  <script type="text/javascript" src="example.js">
  </script>
</head>
<body>
</body>
</html>
```


You can copy any of the examples in this chapter and write them into `example.js`. None of the examples are going to be particularly exciting, but they may be illuminating.

In later chapters, I'll be showing you how to use JavaScript to alter the behavior and content of your document. For now, I'll be using simple dialog boxes to display messages.

Whenever you change the contents of `example.js`, you can test its effects by reloading `test.html` in a web browser. The web browser will interpret the JavaScript code immediately.

Programming languages are either interpreted or compiled. Languages like Java or C++ require a **compiler**. A compiler is a program that translates the source code written in a high-level language like Java into a file that can be executed directly by a computer.

Interpreted languages don't require a compiler—they just need an interpreter instead. With JavaScript, in the context of the World Wide Web, the web browser does the interpreting. The JavaScript interpreter in the browser executes the code directly from the source. Without the interpreter, the JavaScript code would never get executed.

If there are any errors in the code written in a compiled language, those errors will pop up when the code is compiled. In the case of an interpreted language, errors won't become apparent until the interpreter executes the code.

Although compiled languages tend to be faster and more portable than interpreted languages, they often have a fairly steep learning curve.

One of the nice things about JavaScript is that it's relatively easy to pick up. Don't let that fool you though: JavaScript is capable of some pretty complex programming operations. For now, let's take a look at the basics.

Syntax

English is an interpreted language. By reading and processing these words that I have written in English, you are acting as the interpreter. As long as I follow the grammatical rules of English, my writing can be interpreted correctly. These grammatical rules include structural rules known as **syntax**.

Every programming language, just like every written language, has its own syntax. JavaScript has a syntax that is very similar to that of other programming languages like Java and C++.

Statements

A script written in JavaScript, or any other programming language, consists of a series of instructions. These are called **statements**. These statements must be written with the right syntax in order for them to be interpreted correctly.

Statements in JavaScript are like sentences in English. They are the building blocks of any script.

Whereas English grammar demands that sentences begin with a capital letter and end with a period, the syntax of JavaScript is much more forgiving. You can simply separate statements by placing them on different lines:

```
first statement
second statement
```

If you place a number of statements on the same line, you must separate them with semicolons like this:

```
first statement; second statement;
```

However, it is good programming practice to place a semicolon at the end of every statement even if they are on different lines:

```
first statement;
second statement;
```

This helps to make your code more readable. Putting each statement on its own line makes it easier to follow the sequence that your JavaScript is executed in.

Comments

Not all statements are (or need to be) executed by the JavaScript interpreter. Sometimes you'll want to write something purely for your own benefit, and you'll want these statements to be ignored by the JavaScript interpreter. These are called **comments**.

Comments can be very useful when you want to keep track of the flow of your code. They act like sticky notes, helping you to keep track of what is happening in your script.

JavaScript allows you to indicate a comment in a number of different ways. For example, if you begin a line with two forward slashes, that line will be treated as a comment:

```
//  Note to self: comments are good.
```

If you use this notation, you must put the slashes at the start of each comment line. This won't work, for instance:

```
//  Note to self:
    comments are good.
```

Instead, you'd need to write

```
//  Note to self:
//  comments are good.
```

If you want to comment out multiple lines like that, you can place a forward slash and an asterisk at the start of the comment block and an asterisk and forward slash at the end:

```
/*  Note to self:
    comments are good  */
```

This is useful when you need to insert a long comment that will be more readable when it is spread over many lines.

You can also use HTML-style comments, but only for single lines. In other words, JavaScript treats <!– the same way that it treats //:

```
<!– This is a comment in JavaScript.
```

In HTML, you would need to close the comment with –>:

```
<!– This is a comment in HTML –>
```

JavaScript would simply ignore the closing of the comment, treating it as part of the comment itself.

Whereas HTML allows you to split comments like this over multiple lines, JavaScript requires the comment identifier to be at the start of each line.

Because of the confusing differences in how this style of comment is treated by JavaScript, I don't recommend using HTML-style comments. Stick to using two forward slashes for single-line comments and the slash-asterisk notation for multi-line comments.

Variables

In our everyday lives there are some things about us that are fixed and some things that are changeable. My name and my birthday are fixed. My mood and my age, on the other hand, will change over time. The things that are subject to change are called **variables**.

My mood changes depending on how I'm feeling. Suppose I had a variable with the name `mood`. I could use this variable to store my current state of mind. Regardless of whether this variable has the value "happy" or "sad", the name of the variable remains the same: `mood`. I can change the value as often as I like.

Likewise, my age might currently be 33. In one year's time, my age will be 34. I could use a variable named `age` to store how old I am and then update `age` on my birthday. When I refer to `age` now, it has the value 33. In one year's time, the same term will have the value 34.

Giving a value to a variable is called **assignment**. I am assigning the value "happy" to the variable `mood`. I am assigning the value 33 to the variable `age`.

This is how you would assign these variables in JavaScript:

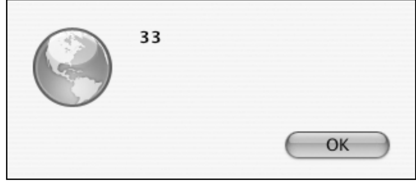
```
mood = "happy";
age = 33;
```

When a variable has been assigned a value, we say that the variable **contains** the value. The variable `mood` now contains the value "happy". The variable `age` now contains the value 33. You could then display the values of these two variables in annoying pop-up `alert` windows by using the statements

```
alert(mood);
alert(age);
```

Here is an example of the value of the variable called `mood`:

Here is an example of the value of the variable called `age`:



We'll get on to doing useful things with variables later on in the book, don't you worry!

Notice that you can jump right in and start assigning values to variables without introducing them first. In many programming languages, this isn't allowed. Other languages demand that you first introduce, or **declare**, any variables.

In JavaScript, if you assign a value to a variable that hasn't yet been declared, the variable is declared automatically. Although declaring variables beforehand isn't required in JavaScript, it's still good programming practice. Here's how you would declare `mood` and `age`:

```
var mood;
var age;
```

You don't have to declare variables separately. You can declare multiple variables at the same time:

```
var mood, age;
```

You can even kill two birds with one stone by declaring a variable and assigning it a value at the same time:

```
var mood = "happy";
var age = 33;
```

You could even do this:

```
var mood = "happy", age = 33;
```

That's the most efficient way to declare and assign variables. It has exactly the same meaning as doing this:

```
var mood, age;
mood = "happy";
age = 33;
```

The names of variables, along with just about everything else in JavaScript, are case-sensitive. The variable `mood` is not the same variable as `Mood`, `MOOD` or `mOOd`. These statements would assign values to two different variables:

```
var mood = "happy";
MOOD = "sad";
```

The syntax of JavaScript does not allow variable names to contain spaces or punctuation characters (except for the dollar symbol, $). The next line would produce a syntax error:

```
var my mood = "happy";
```

Variable names can contain letters, numbers, dollar symbols, and underscores. In order to avoid long variables looking all squashed together, and to improve readability, you can use underscores in variable names:

```
var my_mood = "happy";
```

The text "happy" in that line is an example of a **literal**. A literal is something that is literally written out in the JavaScript code. Whereas the word `var` is a keyword and `my_mood` is the name of a variable, the text "happy" doesn't represent anything other than itself. To paraphrase Popeye, "It is what it is!"

Data types

The value of `mood` is a *string literal*, whereas the value of age is a *number literal*. These are two different types of data, but JavaScript makes no distinction in how they are declared or assigned. Some other languages demand that when a variable is declared, its data type is also declared. This is called **typing**.

Programming languages that require explicit typing are called **strongly typed** languages. Because typing is not required in JavaScript, it is a **weakly typed** language. This means that you can change the data type of a variable at any stage.

The following statements would be illegal in a strongly typed language but are perfectly fine in JavaScript:

```
var age = "thirty three";
age = 33;
```

JavaScript doesn't care whether age is a *string* or a *number*.

Now let's review the most important data types that exist within JavaScript.

Strings

Strings consist of zero or more characters. Characters include letters, numbers, punctuation marks, and spaces. Strings must be enclosed in quotes. You can use either single quotes or double quotes. Both of these statements have the same result:

```
var mood = 'happy';
var mood = "happy";
```

Use whichever one you like, but it's worth thinking about what characters are going to be contained in your string. If your string contains the double-quote character, then it makes sense to use single quotes to enclose the string. If the single-quote character is part of the string, you should probably use double quotes to enclose the string:

```
var mood = "don't ask";
```

If you wanted to write that statement with single quotes, you would need to ensure that the apostrophe (or single quote) between the `n` and the `t` is treated as part of the string. In this case, the single quote needs to be treated the same as any other character, rather than as a signal for marking the end of the string. This is called **escaping**. In JavaScript, escaping is done using the backslash character:

```
var mood = 'don\'t ask';
```

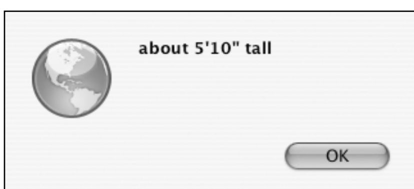
Similarly, if you enclose a string with double quotes, but that string also contains a double-quote character, you can use the backslash to escape the double-quote character within the string:

```
var height = "about 5'10\" tall";
```

These backslashes don't actually form part of the string. You can test this for yourself by adding this to your `example.js` file and reloading `test.html`:


```
var height = "about 5'10\" tall";
alert(height);
```

Here's an example of an output of a variable using backslashes to escape characters:



Personally, I like to use double quotes. Whether you decide to use double or single quotes, it's best to be consistent. If you switch between using double and single quotes all the time, your code could quickly become hard to read.

Numbers

If you want a variable to contain a numeric value, you don't have to limit yourself to whole numbers. JavaScript also allows you to specify numbers to as many decimal places as you want. These are called **floating-point numbers**:

```
var age = 33.25;
```

You can also use negative numbers. A minus sign at the beginning of a number indicates that it's negative:

```
var temperature = -20;
```

Negative values aren't limited to whole numbers either:

```
var temperature = -20.33333333
```

These are all examples of the *number* data type.

Boolean values

Another data type is *Boolean*.

Boolean data has just two possible values: true or false. Let's say I wanted a variable to store one value for when I'm sleeping and another value for when I'm not sleeping. I could use the string data type and assign it values like "sleeping" or "not sleeping", but it makes much more sense to use the Boolean data type:

```
var sleeping = true;
```

Boolean values lie at the heart of all computer programming. At a fundamental level, all electrical circuits use only Boolean data: either the current is flowing or it isn't. Whether you think of it in terms of "true and false", "yes and no", or "one and zero", the important thing is that there can only ever be one of two values.

Boolean values, unlike string values, are not enclosed in quotes. There is a difference between the Boolean value false and the string value "false".

This will set the variable `married` to the Boolean value true:

```
var married = true;
```

In this case, `married` is a string containing the word "true":

```
var married = "true";
```

Arrays

Strings, numbers, and Boolean values are all examples of **scalars**. If a variable is a scalar, then the variable can only ever have one value at any one time. If you want to use a variable to store a whole set of values, then you need an **array**.

An array is a grouping of multiple values under the same name. Each one of these values is an **element** of the array. For instance, you might want to have a variable called `beatles` that contains the names of all four members of the band at once.

In JavaScript, you declare an array by using the `Array` keyword. You can also specify the number of elements that you want the array to contain. This number is the **length** of the array:

```
var beatles = Array(4);
```

Sometimes you won't know in advance how many elements an array is eventually going to hold. That's OK. Specifying the number of elements is optional. You can just declare an array with an unspecified number of elements:

```
var beatles = Array();
```

Adding elements to an array is called **populating**. When you populate an array, you specify not just the value of the element, but also where the element comes in the array. This is the **index** of the element. Each element has a corresponding index. The index is contained in square brackets:

```
array[index] = element;
```

Let's start populating our array of Beatles. We'll go in the traditional order of John, Paul, George, and Ringo. Here's the first index and element:

```
beatles[0] = "John";
```

I know it might seem counterintuitive to start with an index of zero instead of one, but I'm afraid that's just the way that JavaScript works. It's easy to forget this. Many novice programmers have fallen into this common pitfall when first using arrays.

Here's how we'd declare and populate our entire `beatles` array:

```
var beatles = Array(4);
beatles[0] = "John";
beatles[1] = "Paul";
beatles[2] = "George";
beatles[3] = "Ringo";
```

You can now retrieve the element "George" in your script by referencing the index 2 (`beatles[2]`). It might take a while to get used to the fact that the length of the array is four when the last element has an index of three. That's an unfortunate result of arrays beginning with the index number zero.

That was a fairly long-winded way of populating an array. You can take a shortcut by populating your array at the same time that you declare it. When you are populating an array in a declaration, separate the values with commas:

```
var beatles = Array("John","Paul","George","Ringo");
```

An index will automatically be assigned for each element. The first index will be zero, the next will be one, etc. So referencing `beatles[2]` will still give us "George".

You don't even have to specify that you are creating an array. Instead, you can use square brackets to group the initial values together:

```
var beatles = ["John","Paul","George","Ringo"];
```

Still, it's good to get into the habit of using the `Array` keyword when you declare or populate an array. Your scripts will be more readable and it will be easy to spot arrays at a glance.

The elements of an array don't have to be strings. You can store Boolean values in an array. You can also use an array to store a series of numbers:

```
var years = Array(1940,1941,1942,1943);
```

You can even use a mixture of all three:

```
var lennon = Array("John",1940,false);
```

An element can be a variable:

```
var name = "John";
beatles[0] = name;
```

This would assign the value "John" to the first element of the `beatles` array.

The value of an element in one array can be an element from another array. This will assign the value "Paul" to the second element of the `beatles` array:

```
var names = Array("Ringo","John","George","Paul");
beatles[1] = names[3];
```

In fact, arrays can hold other arrays! Any element of an array can contain an array as its value:

```
var lennon = Array("John",1940,false);
var beatles = Array();
beatles[0] = lennon;
```

Now the value of the first element of the `beatles` array is itself an array. To get the values of each element of this array, we need to use some more square brackets. The value of `beatles[0][0]` is "John", the value of `beatles[0][1]` is 1940 and the value of `beatles[0][2]` is false.

This is quite a powerful way of storing and retrieving information, but it's going to be a frustrating experience if we have to remember the numbers for each index (especially when we have to start counting from zero). Luckily, there is a far more readable way of populating arrays.

Associative arrays

The `beatles` array is an example of a **numeric array**. The index for each element is a number that increments with each addition to the array. The index of the first element is zero, the index of the second element is one, and so on.

If you only specify the values of an array, then that array will be numeric. The index for each element is created and updated automatically.

It is possible to override this behavior by specifying the index of each element. When you specify the index, you don't have to limit yourself to numbers. The index can be a string instead:

```
var lennon = Array();
lennon["name"] = "John";
lennon["year"] = 1940;
lennon["living"] = false;
```

This is called an **associative array**. Actually, all arrays are associative arrays when you think about it. It just so happens that each index of a numeric array is created automatically. Each index is still associated with a specific value. So a numeric array is really just another example of an associative array.

Using associative instead of numeric arrays means you can reference elements by name instead of relying on numbers. It also makes for more readable scripts.

Let's create a new array named beatles and populate one of its elements with the array lennon that we created previously. Remember, an element in an array can itself be an array:

```
var beatles = Array();
beatles[0] = lennon;
```

Now we can get at the elements we want without using any numbers. beatles[0]["name"] is "John", beatles[0]["year"] is 1940, and beatles[0]["living"] is false.

That's an improvement, but we can go one further. What if beatles was an associative array instead of a numerical array? Then, instead of using numbers to reference each element of the array, we could use descriptive strings like "drummer" or "bassist":

```
var beatles = Array();
beatles["vocalist"] = lennon;
```

Now the value of beatles["vocalist"]["name"] is "John", beatles["vocalist"]["year"] is 1940, and beatles["vocalist"]["living"] is false.

Operations

All the statements I've shown you have been very simple. All I've done is create different types of variables. In order to do anything useful with JavaScript, we need to be able to do calculations and manipulate data. We want to perform **operations**.

Arithmetic operators

Addition is an operation. So are subtraction, division, and multiplication. Every one of these **arithmetic operations** requires an **operator**. Operators are symbols that JavaScript has reserved for performing operations. You've already seen one operator in action. We've been using the equals sign (=) to perform assignment. The operator for addition is the plus sign (+), the operator for subtraction is the minus sign (-), division uses the forward slash (/), and the asterisk (*) is the symbol for multiplication operations.

Here's a simple addition operation:

```
1 + 4
```

You can also combine operations:

```
1 + 4 * 5
```

To avoid ambiguity, it's best to separate operations by enclosing them in parentheses:

```
1 + (4 * 5)
(1 + 4) * 5
```

A variable can contain an operation:

```
var total = (1 + 4) * 5;
```

Best of all, you can perform operations on variables:

```
var temp_fahrenheit = 95;
var temp_celsius = (temp_fahrenheit - 32) / 1.8;
```

JavaScript provides some useful operators that act as shortcuts in frequently used operations. If you wanted to increase the value of a numeric variable by one, you could write

```
year = year + 1;
```

You can achieve the same result by using the ++ operator:

```
year++;
```

Similarly, the `--` operator will decrease the value of a numeric variable by one.

The + operator is a bit special. You can use it on strings as well as numbers. Joining strings together is a straightforward operation:

```
var message = "I am feeling " + "happy";
```

Joining strings together like this is called **concatenation**. This also works on variables:

```
var mood = "happy";
var message = "I am feeling " + mood;
```

You can even concatenate numbers with strings. This is possible because of JavaScript's weakly typed nature. The number will automatically be converted to a string:

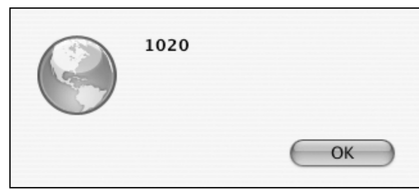
```
var year = 2005;
var message = "The year is " + year;
```

Remember, if you concatenate a string with a number, the result will be a longer string, but if you use the same operator on two numbers, the result will be the sum of the two numbers. Compare the results of these two `alert` statements:

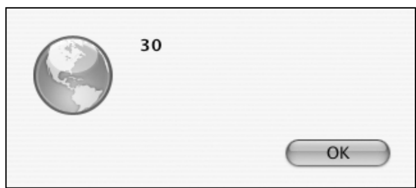
```
alert ("10" + 20);
alert (10 + 20);
```

The first `alert` returns the string "1020". The second returns the number 30.

Here's the result of concatenating the string "10" and the number 20:



The result of adding the number 10 and the number 20 is as follows:



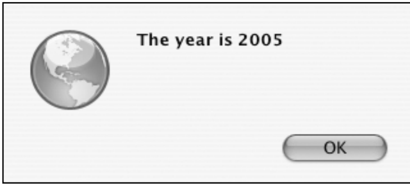
Another useful shorthand operator is += which performs addition and assignment (or concatenation and assignment) at the same time:

```
var year = 2005;
var message = "The year is ";
message += year;
```

The value of `message` is now "The year is 2005". You can test this yourself by using another `alert` dialog box:

```
alert(message);
```

The result of concatenating a string and a number is as follows:



Conditional statements

All the statements you've seen so far have been relatively simple declarations or operations. The real power of a script is its ability to make decisions based on the criteria it is given. JavaScript makes those decisions by using **conditional statements**.

When a browser is interpreting a script, statements are executed one after another. You can use a conditional statement to set up a condition that must be successfully evaluated before more statements are executed. The most common conditional statement is the `if` statement. It works like this:

```
if (condition) {
  statements;
}
```

The condition is contained within parentheses. The condition always resolves to a Boolean value, which is either true or false. The statement or statements contained within the curly braces will only be executed if the result of the condition is true. In this example, the annoying `alert` message never appears:

```
if (1 > 2) {
  alert("The world has gone mad!");
}
```

The result of the condition is false because one is not greater than two.

I've indented everything between the curly braces. This is not a syntax requirement of JavaScript—I've done it purely to make my code more readable.

In fact, the curly braces themselves aren't completely necessary. If you only want to execute a single statement based on the outcome of an `if` statement, you don't have to use curly braces at all. You can just put everything on one line:

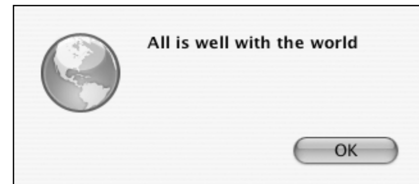
```
if (1 > 2) alert("The world has gone mad!");
```

However, the curly braces help make scripts more readable so it's a good idea to use them anyway.

The `if` statement can be extended using `else`. Statements contained in the `else` clause will only be executed when the condition is false:

```
if (1 > 2) {
  alert("The world has gone mad!");
} else {
  alert("All is well with the world");
}
```

This is returned when 1>2 is false:



Comparison operators

JavaScript provides plenty of operators that are used almost exclusively in conditional statements. There are comparison operators like greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

If you want to find out if two values are equal, you can use the equality operator. It consists of two equals signs (==). Remember, a single equals sign is used for assignment. If you use a single equals sign in a conditional statement, the operation will always be true as long as the assignment succeeds.

This is the *wrong* way to check for equality:

```
var my_mood = "happy";
var your_mood = "sad";
if (my_mood = your_mood) {
  alert("We both feel the same.");
}
```

I've just assigned the value of your_mood to my_mood. The assignment operation was carried out successfully so the result of the conditional statement is true.

This is what I should have done:

```
var my_mood = "happy";
var your_mood = "sad";
if (my_mood == your_mood) {
 alert("We both feel the same.");
}
```

This time, the result of the conditional statement is false.

There is also an operator that tests for inequality. Use an exclamation point followed by an equals sign (!=).

```
if (my_mood != your_mood) {
  alert("We're feeling different moods.");
}
```

Logical operators

It's possible to combine operations in a conditional statement. Say I want to find out if a certain variable, let's call it num, has a value between five and ten. I need to perform two operations. First, I need to find out if the variable is greater than or equal to five, and next I need to find out if the variable is less than or equal to ten. These operations are called **operands**. This is how I combine operands:

```
if ( num>=5 && num<=10 ) {
  alert("The number is in the right range.");
}
```

I've used the "and" operator, represented by two ampersands (&&). This is an example of a **logical operator**.

Logical operators work on Boolean values. Each operand returns a Boolean value of either true or false. The "and" operation will be true only if both operands are true.

The logical operator for "or" is two vertical pipe symbols (||). The "or" operation will be true if one of its operands is true. It will also be true if both of its operands are true. It will be false only if both operands are false.

```
if ( num > 10 || num < 5 ) {
  alert("The number is not in the right range.");
}
```

There is one other logical operator. It is represented by a single exclamation point (!). This is the "not" operator. The "not" operator works on just a single operand. Whatever Boolean value is returned by that operand gets reversed. If the operand is true, the "not" operator switches it to false:

```
if ( !(1 > 2) ) {
  alert("All is well with the world");
}
```

Notice that I've placed the operand in parentheses to avoid any ambiguities. I want the "not" operator to act on everything between the parentheses.

You can use the "not" operator on the result of a complete conditional statement to reverse its value. I'm going to use another set of parentheses so that the "not" operator works on both operands combined:

```
if ( !(num > 10 || num < 5) ) {
  alert("The number IS in the right range.");
}
```

Looping statements

The `if` statement is probably the most important and useful conditional statement. The only drawback to the `if` statement is that it can't be used for repetitive tasks. The block of code contained within the curly braces is executed once. If you want to execute the same code a number of times, you'll need to use a looping statement.

Looping statements allow you to keep executing the same piece of code over and over. There are a number of different types of looping statements, but they all work in much the same way. The code within a looping statement continues to be executed as long as the condition is met. When the condition is no longer true, the loop stops.

while

The while loop is very similar to the if statement. The syntax is the same:

```
while (condition) {
  statements;
}
```


The only difference is that the code contained within the curly braces will be executed over and over as long as the condition is true. Here's an example of a while loop:

```
var count = 1;
while (count < 11) {
  alert (count);
  count++;
}
```

Let's take a closer look at the code I just showed you. I began by creating a numeric variable, count, containing the value one. Then I created a while loop with the condition that the loop should repeat as long as the value of count is less than eleven. Inside the loop itself, the value of count is incremented by one using the ++ operator. The loop will execute ten times. In your web browser, you will see an annoying alert dialog box flash up ten times. After the loop has been executed, the value of count will be eleven.

> *It's important that something happens within the* while *loop that will affect the test condition. In this case, we increase the value of* count *within the* while *loop. This results in the condition evaluating to false after ten loops. If we didn't increase the value of the* count *variable, the* while *loop would execute forever.*

do...while

As with the if statement, it is possible that the statements contained within the curly braces of a while loop may never be executed. If the condition evaluates as false on the first loop, then the code won't be executed even once.

There are times when you will want the code contained within a loop to be executed at least once. In this case, it's best to use a do loop. This is the syntax for a do loop:

```
do {
  statements;
} while (condition);
```

This is very similar to the syntax for a regular while loop, but with a subtle difference. Even if the condition evaluates as false on the very first loop, the statements contained within the curly braces will still be executed once.

Let's look at our previous example, reformatted as a do...while loop:

```
var count = 1;
do {
  alert (count);
  count++;
} while (count < 11);
```

The result is exactly the same as the result from our while loop. The alert message appears ten times. After the loop is finished, the value of the variable count is eleven.

Now consider this variation:

```
var count = 1;
do {
  alert (count);
  count++;
} while (count < 1);
```

In this case, the condition never evaluates as true. The value of count is one to begin with so it is never less than one. Yet the do loop is still executed once because the condition comes after the curly braces. You will still see one alert message. After these statements are executed, the value of count is two even though the condition is false.

for

The for loop is a convenient way of executing some code a specific number of times. In that sense, it's similar to the while loop. In a way, the for loop is just a reformulation of the do loop we've already used. If we look at our do loop example, we can formulate it in full like this:

```
initialize;
while (condition) {
  statements;
  increment;
}
```

The for loop simply reformulates that as follows:

```
for (initial condition; test condition; alter condition) {
  statements;
}
```

This is generally a cleaner way of executing loops. Everything relevant to the loop is contained within the parentheses of the for statement.

If we reformulate our do loop example, this is how it looks:

```
for (var count = 1; count < 11; count++ ) {
  alert (count);
}
```

Everything related to the loop is contained within the parentheses. Now we can put code between the curly braces, secure in the knowledge that the code will be executed exactly ten times.

One of the most common uses of the `for` loop is to act on every element of an array. This is achieved using `array.length`, which provides the number of elements in array:

```
var beatles = Array("John","Paul","George","Ringo");
for (var count = 0 ; count < beatles.length; count++ ) {
  alert(beatles[count]);
}
```

If you run this code, you will see four `alert` messages, one for each Beatle.

Functions

If you want to re-use the same piece of code more than once, you can wrap the statements up inside a **function**. A function is a group of statements that can be invoked from anywhere in your code. Functions are, in effect, miniature scripts.

It's good practice to define your functions before you invoke them.

A simple function might look like this:

```
function shout() {
  var beatles = Array("John","Paul","George","Ringo");
  for (var count = 0 ; count < beatles.length; count++ ) {
    alert(beatles[count]);
  }
}
```

This function performs the loop that pops up the names of each Beatle. Now, whenever you want that action to occur later in your script, you can invoke the function by simply writing

```
shout();
```

That's a useful way of avoiding lots of typing whenever you want to carry out the same action more than once. The real power of functions is that you can pass data to them and then have them act on that data. When data is passed to a function, it is known as an **argument**.

Here's the syntax for defining a function:

```
function name(arguments) {
  statements;
}
```

JavaScript comes with a number of built-in functions. You've seen one of them already: the `alert` function takes a single argument and then pops up a dialog box with the value of the argument.

You can define a function to take as many arguments as you want by separating them with commas. Any arguments that are passed to a function can be used just like regular variables within the function.

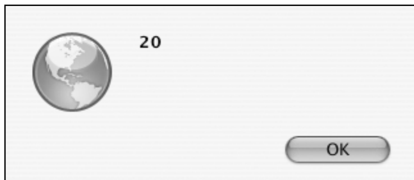
Here's a function that takes two arguments. If you pass this function two numbers, the function will multiply them:

```
function multiply(num1,num2) {
  var total = num1 * num2;
  alert(total);
}
```

You can invoke the function from anywhere in your script, like this:

```
multiply(10,2);
```

The result of passing the values 10 and 2 to the `multiply()` function is as follows:



This will have the effect of immediately popping up an `alert` dialog with the answer (20). It would be much more useful if the function could send the answer back to the statement that invoked the function. This is quite easily done. As well as accepting data (in the form of arguments), functions can also return data.

You can create a function that returns a number, a string, an array, or a Boolean value. Use the **return** statement to do this:

```
function multiply(num1,num2) {
  var total = num1 * num2;
  return total;
}
```

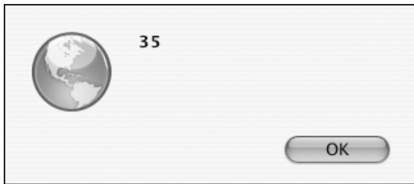
Here's a function that takes one argument (a temperature in degrees Fahrenheit) and returns a number (the same temperature in degrees Celsius):

```
function convertToCelsius(temp) {
  var result = temp - 32;
  result = result / 1.8;
  return result;
}
```

The really useful thing about functions is that they can be used as a data type. You can assign the result of a function to a variable:

```
var temp_fahrenheit = 95;
var temp_celsius = convertToCelsius(temp_fahrenheit);
alert(temp_celsius);
```


The result of converting 95 degrees Fahrenheit into Celsius is as follows:



In this example, the variable `temp_celsius` now has a value of 35, which was returned by the `convertToCelsius` function.

You might be wondering about the way I've named my variables and functions. For my variables, I've used underscores to separate words. For my functions, I've used capital letters after the first word (this is called **camel case**). I've done this purely for my own benefit so that I can easily distinguish between variables and functions. As with variables, function names cannot contain spaces. Camel casing is simply a convenient way to work within that restriction.

Variable scope

I've mentioned already that it's good programming practice to use `var` when you are assigning a value to a variable for the first time. This is especially true when you are using variables in functions.

A variable can be either global or local. When we differentiate between local and global variables, we are discussing the **scope** of variables.

A **global variable** can be referenced from anywhere in the script. Once a global variable has been declared in a script, that variable can be accessed from anywhere in that script, even within functions. Its scope is global.

A **local variable** exists only within the function in which it is declared. You can't access the variable outside the function. It has a local scope.

So, you can use both global and local variables within functions. This can be useful, but it can also cause a lot of problems. If you unintentionally use the name of a global variable within a function, JavaScript will assume that you are referring to the global variable, even if you actually intended the variable to be local.

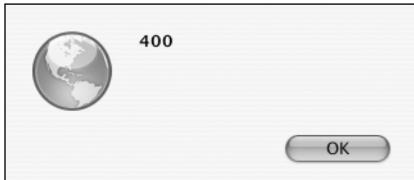
Fortunately, you can use the `var` keyword to explicitly set the scope of a variable within a function.

If you use var within a function, the variable will be treated as a local variable. It only exists within the context of the function. If you don't use var, the variable will be treated as a global variable. If there is already a variable with that name, the function will overwrite its value.

Take a look at this example:

```
function square(num) {
  total = num * num;
  return total;
}
var total = 50;
var number = square(20);
alert(total);
```

The value of the variable has been inadvertently changed:



The value of the variable total is now 400. All I wanted from the square() function was for it to return the value of number squared. But because I didn't explicitly declare that the variable called total within the function should be local, the function has changed the value of the global variable called total.

This is how I should have written the function:

```
function square(num) {
  var total = num * num;
  return total;
}
```

Now I can safely have a global variable named total, secure in the knowledge that it won't be affected whenever the square() function is invoked.

Remember, functions should behave like self-contained scripts. That's why you should always declare variables within functions as being local in scope. If you always use var within functions, you can avoid any potential ambiguities.

Objects

There is one very important data type that we haven't looked at yet: **objects**. An object is a self-contained collection of data. This data comes in two forms: **properties** and **methods**:

- A property is a variable belonging to an object.
- A method is a function that the object can invoke.

These properties and methods are all combined in one single entity, which is the object.

Properties and methods are both accessed in the same way using JavaScript's dot syntax:

```
Object.property
Object.method()
```


You've already seen how variables can be used to hold values for things like mood and age. If there were an object called, say, Person, then these would be properties of the object:

```
Person.mood
Person.age
```

If there were functions associated with the Person object—say, walk() or sleep()—then these would be methods of the object:

```
Person.walk()
Person.sleep()
```

Now all these properties and methods are grouped together under one term: Person.

To use the Person object to describe a specific person, you would create an **instance** of Person. An instance is an individual example of a generic object. For instance, you and I are both people, but we are also both individuals. We probably have different properties (our ages may differ, for instance), yet we are both examples of an object called Person.

A new instance is created using the new keyword:

```
var jeremy = new Person;
```

This would create a new instance of the object Person, called jeremy. I could use the properties of the Person object to retrieve information about jeremy:

```
jeremy.age
jeremy.mood
```

I've used the imaginary example of a Person object just to demonstrate objects, properties, methods, and instances. In JavaScript, there is no Person object. It is possible for you to create your own objects in JavaScript. These are called **user-defined objects**. But that's quite an advanced subject that we don't need to deal with for now.

Fortunately, JavaScript is like one of those TV chefs who produce perfectly formed creations from the oven, declaring, "Here's one I made earlier." JavaScript comes with a range of pre-made objects that you can use in your scripts. These are called **native objects**.

Native objects

You've already seen objects in action. Array is an object. Whenever you initialize an array using the new keyword, you are creating a new instance of the Array object:

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

When you want to find out how many elements are in an array, you do so by using the length property:

```
beatles.length;
```

The Array object is an example of a native object supplied by JavaScript. Other examples include Math and Date, both of which have very useful methods for dealing with numbers and dates respectively. For instance, the Math object has a method called round which can be used to round up a decimal number:

```
var num = 7.561;
var num = Math.round(num);
alert(num);
```

The Date object can be used to store and retrieve information about a specific date and time. If you create a new instance of the Date object, it will be automatically be pre-filled with the current date and time:

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

The date object has a whole range of methods like getDay(), getHours(), and getMonth() that can be used to retrieve information about the specified date. getDay(), for instance, will return the day of the week of the specified date:

```
var today = current_date.getDay();
```

Native objects like this provide invaluable shortcuts when you're writing JavaScript.

Host objects

Native objects aren't the only kind of pre-made objects that you can use in your scripts. Another kind of object is supplied not by the JavaScript language itself, but by the environment in which it's running. In the case of the Web, that environment is the web browser. Objects that are supplied by the web browser are called **host objects**.

Host objects include Form, Image, and Element. These objects can be used to get information about forms, images, and form elements within a web page.

I'm not going to show you any examples of how to use those host objects. There is another object that can be used to get information about any element in a web page that you might be interested in: the document object. For the rest of this book, we are going to be looking at lots of properties and methods belonging to the document object.

What's next?

In this chapter, I've shown you the basics of the JavaScript language. Throughout the rest of the book, I'll be using terms that have been introduced here: statements, variables, arrays, functions, and so on. Some of these concepts will become clearer once you see them in action in a working script. You can always refer back to this chapter whenever you need a reminder of what these terms mean.

I've just introduced the concept of objects. Don't worry if it isn't completely clear to you just yet. The next chapter will take an in-depth look at one particular object, the `document` object. I want to start by showing you some properties and methods associated with this object. These properties and methods are provided courtesy of the Document Object Model.

In the next chapter, I want to introduce you to the idea of the DOM and show you how to use some of its very powerful methods.