CHAPTER 1

A Brief History of JavaScript

What this chapter covers:

- The origins of JavaScript
- The browser wars
- The evolution of the DOM

When the first edition of this book was published in 2005, it was an exciting time to be a web designer. Thankfully, five years later, it still is. This is especially true for JavaScript, which has been pulled from the shadows and into the spotlight. Web development has evolved from its chaotic, haphazard roots into a mature discipline. Designers and developers are adopting a standards-based approach to building websites, and the term *web standards* has been coined to describe the technologies that enable this approach.

Whenever designers discuss the subject of web standards, Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) usually take center stage. However, a third technology has been approved by the World Wide Web Consortium (W3C) and is supported by all standards-compliant web browsers. This is the Document Object Model (DOM), which allows us to add interactivity to our documents in much the same way that CSS allow us to add styles.

Before looking at the DOM, let's examine the language that you'll be using to make your web pages interactive. The language is JavaScript, and it has been around for quite some time.

The origins of JavaScript

JavaScript was developed by Netscape, in collaboration with Sun Microsystems. Before JavaScript, web browsers were fairly basic pieces of software capable of displaying hypertext documents. JavaScript was later introduced to add some extra spice to web pages and to make them more interactive. The first version, JavaScript 1.0, debuted in Netscape Navigator 2 in 1995.

At the time of JavaScript 1.0's release, Netscape Navigator dominated the browser market. Microsoft was struggling to catch up with its own browser, Internet Explorer, and was quick to follow Netscape's lead by releasing its own VBScript language, along with a version of JavaScript called JScript, with the delivery of Internet Explorer 3. As a response to this, Netscape and Sun, together with the European Computer Manufacturers Association (ECMA), set about standardizing the language. The result was ECMAScript, yet another name for the same language. Though the name never really stuck, we should really be referring to JavaScript as ECMAScript.

JavaScript, ECMAScript, JScript—whatever you want to call it—was gaining ground by 1996. Version 3 browsers from Netscape and Microsoft both supported the JavaScript 1.1 language to varying degrees.

■ **Note** JavaScript has nothing to do with Java, a programming language developed by Sun Microsystems. JavaScript was originally going to be called LiveScript. JavaScript was probably chosen to make the new language sound like it was in good company. Unfortunately, the choice of this name had the effect of confusing the two languages in people's minds—a confusion that was amplified by the fact that web browsers also supported a form of client-side Java. However, while Java's strength lies in the fact that it can theoretically be deployed in almost any environment, JavaScript was always intended for the confines of the web browser.

JavaScript is a scripting language. Unlike a program that does everything itself, the JavaScript language simply tells the web browser what to do. The web browser interprets the script and does all the work, which is why JavaScript is often compared unfavorably with compiled programming languages like Java and C++. But JavaScript's relative simplicity is also its strength. Because it has a low barrier to entry, nonprogrammers who wanted to cut and paste scripts into their existing web pages quickly adopted the language.

JavaScript also offers developers the chance to manipulate aspects of the web browser. For example, the language could be used to adjust the properties of a browser window, such as its height, width, and position. Addressing the browser's own properties in this way can be thought of as a Browser Object Model. Early versions of JavaScript also provided a primitive sort of DOM.

The Document Object Model

What is the DOM? In short, the DOM is a way of conceptualizing the contents of a document.

In the real world, we all share something that could be called a World Object Model. We can refer to objects in our environment using terms like *car*, *house*, and *tree*, and be fairly certain that our terms will be understood. That's because we have mutually agreed on which objects the words refer to specifically. If you say "The car is in the garage," it's safe to assume that the person you're talking to won't take that to mean "The bird is in the cupboard."

Our World Object Model isn't restricted to tangible objects though; it also applies to concepts. For instance, you might refer to "the third house on the left" when giving directions. For that description to make sense, the concepts of "third" and "left" must be understood. If you give that description to someone who can't count, or who can't tell left from right, then the description is essentially meaningless, whether or not the words have been understood. In reality, because people agree on a conceptual World Object Model, very brief descriptions can be full of meaning. You can be fairly sure that others share your concepts of *left* and *third*.

It's the same situation with web pages. Early versions of JavaScript offered developers the ability to query and manipulate some of the actual contents of web documents—mostly images and forms. Because the terms *images* and *forms* had been predefined, JavaScript could be used to address the third image in the document or the form named `details`, as follows:

```
document.images[2]
document.forms['details']
```

This first, tentative sort of DOM is often referred to as DOM Level 0. In those early, carefree days, the most common usage of DOM Level 0 was for image rollovers and some client-side form validation. But when the fourth generation of browsers from Netscape and Microsoft appeared, the DOM really hit the fan.

The browser wars

Netscape Navigator 4 was released in June 1997, and by October of that year, Internet Explorer 4 had also been released. Both browsers promised improvements on previous versions, along with many additions to what could be accomplished with JavaScript, using a greatly expanded DOM. Web designers were encouraged to test-drive the latest buzzword: *DHTML*.

The D word: DHTML

DHTML is short for Dynamic HTML. Not a technology in and of itself, DHTML is a shorthand term for describing the combination of HTML, CSS, and JavaScript. The thinking behind DHTML went like this:

- You could use HTML to mark up your web page into elements.
- You could use CSS to style and position those elements.
- You could use JavaScript to manipulate and change those styles on the fly.

Using DHTML, complex animation effects suddenly became possible. Let's say you used HTML to mark up a page element like this:

```
<div id="myelement">This is my element</div>
```

You could then use CSS to apply positioning styles like this:

```
#myelement {
  position: absolute;
  left: 50px;
  top: 100px;
}
```

Then, using JavaScript, you could change the `left` and `top` styles of `myelement` to move it around on the page. Well, that was the theory anyway.

Unfortunately for developers, the Netscape and Microsoft browsers used different, incompatible DOMs. Although the browser manufacturers were promoting the same ends, they each approached the DOM issue in completely different ways.

Clash of the browsers

The Netscape DOM made use of proprietary elements called *layers*. These layers were given unique IDs and then addressed through JavaScript like this:

```
document.layers['myelement']
```

Meanwhile, the Microsoft DOM would address the same element like this:

```
document.all['myelement']
```

The differences didn't end there. Let's say you wanted to find out the left position of `myelement` and assign it to the variable `xpos`. In Netscape Navigator 4, you would do it like this:

```
var xpos = document.layers['myelement'].left;
```

Here's how you would do the same thing in Internet Explorer 4:

```
var xpos = document.all['myelement'].leftpos;
```

This was clearly a ridiculous situation. Developers needed to double their code to accomplish any sort of DOM scripting. In effect, many scripts were written twice: once for Netscape Navigator and once for Internet Explorer. Convoluted browser sniffing was often required to serve up the correct script.

DHTML promised a world of possibilities, but anyone who actually attempted to use it discovered a world of pain instead. It wasn't long before DHTML became a dirty (buzz)word. The technology quickly garnered a reputation for being both overhyped and overly difficult to implement.

Raising the standard

While the browser manufacturers were busy engaging in their battle for supremacy, and using competing DOMs as weapons in their war, the W3C was quietly putting together a standardized DOM. Fortunately, the browser vendors were able to set aside their mutual animosity. Netscape, Microsoft, and other browser manufacturers worked together with the W3C on the new standard, and DOM Level 1 was completed in October 1998.

Consider the example in the previous section. We have a `<div>` with the ID `myelement`, and we're trying to ascertain the value that has been applied to its left position so that we can store that value as the variable `xpos`. Here's the syntax we would use with the new standardized DOM:

```
var xpos = document.getElementById('myelement').style.left
```

At first glance, that might not appear to be an improvement over the nonstandard, proprietary DOMs. However, the standardized DOM is far more ambitious in its scope.

While the browser manufacturers simply wanted some way to manipulate web pages with JavaScript, the W3C proposed a model that could be used by *any* programming language to manipulate *any* document written in *any* markup language.

Thinking outside the browser

The DOM is what's known as an application programming interface (API). APIs are essentially conventions that have been agreed upon by mutual consent. Real-world equivalents would be things like Morse code, international time zones, and the periodic table of the elements. All of these are standards, and they make it easier for people to communicate and cooperate. In situations where a single convention hasn't been agreed upon, the result is often disastrous. For example, competition between metric and imperial measurements has resulted in at least one failed Mars mission.

In the world of programming, there are many different languages, but there are many similar tasks. That's why APIs are so handy. Once you know the standard, you can apply it in many different environments. The syntax may change depending on the language you're using, but the convention remains the same.

So, while we focus specifically on using the DOM with JavaScript in this book, your new knowledge of the DOM will also be useful if you ever need to parse an XML document using a programming language like PHP or Python.

The W3C defines the DOM as "A platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure, and style of documents." The independence of the standardized DOM, together with its powerful scope, places it head and shoulders above the proprietary DOMs created by the bickering browser manufacturers.

The end of the browser wars

Microsoft won the battle against Netscape for browser market-share supremacy. Ironically, the clash of competing DOMs and proprietary markup had little effect on the final outcome. Internet Explorer was destined to win simply by virtue of the fact that it came preinstalled on all PCs running the Windows operating system.

The people who were hit hardest by the browser wars were web designers. Cross-browser development had become a nightmare. As well as the discrepancies in JavaScript implementations mentioned earlier, the two browsers also had very different levels of support for CSS. Creating style sheets and scripts that worked on both browsers became a kind of black art.

A backlash began against the proprietary stance of the browser manufacturers. A group was formed, calling itself the Web Standards Project, or the WaSP for short (`http://webstandards.org/`). The first task that the WaSP undertook was to encourage browser makers to adopt W3C recommendations—the very same recommendations that the browser manufacturers had helped draft.

Whether it was due to pressure from the WaSP or the result of internal company decisions, there was far greater support for web standards in the next generation of web browsers.

A new beginning

A lot has changed since the early days of the browser wars, and things are still changing almost daily. Some browsers, such as Netscape Navigator, have all but vanished, and new ones have appeared on the scene. When Apple debuted its Safari web browser in 2003 (based on WebKit), there was no question that it would follow the DOM standards. Today, Firefox, Chrome, Opera, Internet Explorer, and a number of different WebKit-based browsers all have excellent support for the DOM. Many of the latest smartphone browsers are using the WebKit rendering engine, pushing browser development forward and making the browser-in-your-pocket superior to some desktop browsers.

■ **Note** WebKit (`http://webkit.org`) is the open source web browser engine use in Safari and Chrome. Open source engines such as WebKit and Gecko (used in Firefox, `https://developer.mozilla.org/en/Gecko`) have played a big role in pushing proprietary browser engines such as Microsoft's Trident (in Internet Explorer) to adopt more progressive web standards.

Today, pretty much all browsers have built-in support for the DOM. The browser wars of the late 1990s appear to be truly behind us. Now it's a race to implement the latest specification first. We've already seen an explosion of DOM scripting with the advent of asynchronous data transfers (Ajax), and the advancements in the HTML5 DOM are adding many new possibilities. HTML5 gives us vastly improved semantics, control over rich media with the `<audio>` and `<video>` elements, the capability of drawing with the `<canvas>` element, local browser storage for more than just cookies, built-in drag-and-drop support, and a lot more.

Life has improved greatly for web designers. Although no single browser has implemented the W3C DOM perfectly, all modern browsers cover about 95% of the specification, and they are each implementing the latest features almost as fast as we can adopt them. This means there's a huge amount that we can accomplish without needing to worry about branching code. Instead of writing scripts with forked code served up with complicated browser sniffing, we are now in a position to write something once and publish it everywhere. As long as we follow the DOM standards, we can be sure that our scripts will work almost universally.

What's next?

As you learned in this brief history lesson, different browsers used to accomplish the same tasks in different ways. This inescapable fact dominated not just the writing of JavaScript scripts, but also how books about JavaScript were written.

Any JavaScript books aimed at demonstrating how to learn the language by example often needed to show the same scripts written in different ways for different browsers. Just like the code found on most websites, the examples in most JavaScript books were full of browser sniffing and code branching. Similarly, technical reference books on JavaScript couldn't simply contain lists of functions and methods. They also needed to document which functions and methods were supported by which browsers.

The situation has changed now. Thanks to the standardization of the DOM, different browsers do the same things in much the same way. This means that when we're talking about how to do something using JavaScript and the DOM, we won't get sidetracked by browser inconsistencies. In fact, as much as possible, this book avoids mentioning any specific browsers.

This book also does not use the term *DHTML*. The term always worked better as a marketing buzzword than as a technical description. For one thing, confusingly, it sounds like another flavor of HTML or XHTML. Also, the term comes with a lot of baggage. If you mention DHTML to anyone who tried using it in the late 1990s, you'll have a hard time convincing them that it's a straightforward, standardized technology now.

DHTML was supposed to refer to the combination of (X)HTML, CSS, and JavaScript, similar to how people are using the term *HTML5* today. In fact, what binds (X)HTML, CSS, and JavaScript together is the DOM. So, let's use a more accurate term to describe this process: *DOM scripting*. This refers to the manipulation of documents and style sheets using the W3C DOM. Whereas DHTML referred only to web documents, DOM scripting can be used in conjunction with any marked-up document using any language that supports the DOM API. In the case of web documents, the ubiquity of JavaScript makes it the best choice for DOM scripting.

Before we get down to the nitty-gritty of DOM scripting, in the next chapter, we'll briefly review JavaScript syntax.

CHAPTER 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 three ways of doing this. You can place the JavaScript between `<script>` tags within the `<head>` of the document:

```
<!DOCTYPE html >
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <title>Example</title>
  <script>
    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. Traditionally you would include this in the `<head>` portion of the document by using the `src` attribute in a `<script>` tag to point to this file:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8"/>
  <title>Example</title>
  <script src="file.js"></script>
</head>
<body>
  Mark-up goes here...
</body>
</html>
```

However, the best technique is to place the `<script>` tag at the end of the document right before the closing `</body>` tag:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <title>Example</title>
</head>
<body>
  Mark-up goes here...
  <script src="file.js"></script>
</body>
</html>
```

Placing the `<script>` tag at the end of the document lets the browser load the page faster (we'll discuss this more in Chapter 5, "Best Practices").

The <script> tags in the preceding examples don't contain the traditional `type="text/javascript"` attribute. This attribute is considered unnecessary as the script is already assumed to be JavaScript.

If you'd like to follow along and 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 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 >
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <title>Just a test</title>
</head>
<body>
  <script src="example.js"></script>
</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, we'll be showing you how to use JavaScript to alter the behavior and content of your document. For now, I'll use 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 be 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 we have written in English, you are acting as the interpreter. As long as we follow the grammatical rules of English, our 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, we 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.

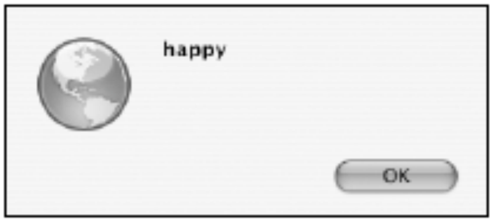
This is how you would assign these variables in JavaScript:

```
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

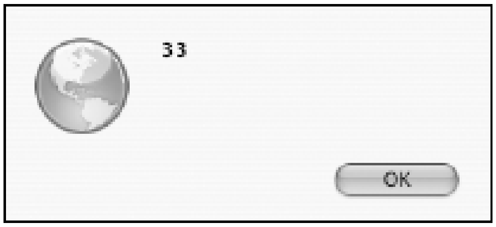
```
alert(mood);
alert(age);
```

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



```
happy
```

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



```
33
```

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";
```

Another stylistic alternative is to use the *camel case* format, where the space is removed and each new word begins with a capital letter:

```
var myMood = "happy";
```

Traditionally the camel case format is preferred for function and method names as well as object properties.

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';
```

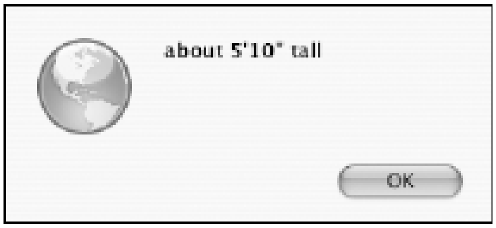
Similarly, if you enclose a string with double quotes, but that string also contains a double-quote character, you can use the backslash to escape the double-quote character within the string:

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

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

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

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


```
about 5'10" tall
```

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 you wanted a variable to store one value for when you're sleeping and another value for when you're no. You 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 "false" as a string.

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 it 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* it. 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";
```

It might seem counterintuitive to start with an index of zero instead of one, but that's just the way 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" ];
```

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 = [ 1940, 1941, 1942, 1943 ];
```

You can even use a mixture of all three:

```
var lennon = [ "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 = [ "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 = [ "John", 1940, false ];
var beatles = [];
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 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 are a couple more ways of storing data. First, we’ll look at a more readable way of populating arrays, and then move on to the preferred method, storing data as an object.

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*. It’s much more readable than a numeric array because you can use strings instead of numbers, but they're actually considered bad form and we recommend that you don’t use them. The reason for this is that when you create an associative array, you're actually creating properties on the Array object. In JavaScript, all variables are really objects of some type. A boolean is a Boolean object. An array is an Array object, and so on. In this example you are giving the lennon array new name, year and living properties. Ideally you don't want to be modifying the properties of the Array object. Instead, you should be using a generic Object.

Objects

Like an array, an *object* is a grouping of multiple values under the same name. Each one of these values is a *property* of the object. For instance, the lennon array in the previous section could be created as an object instead:

```
var lennon = Object();
lennon.name = "John";
lennon.year = 1940;
lennon.living = false;
```

Again, like the Array, an object is declared by using the `Object` keyword, but instead of using square brackets with an index to specify elements, you use dot notation and specify the property name the same way you would on any JavaScript object. For more about Objects, see the section "Objects" at the end of this chapter.

Alternatively you can use the more compact curly-brace {} syntax:

```
{ propertyName:value, propertyName:value }
```

For example, the `lennon` object could also be created like this:

```
var lennon = { name:"John", year:1940, living:false };
```

Property names follow the same naming rules as JavaScript variable names and the value can be any JavaScript value, including other objects.

Using objects 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 `lennon` object that we created previously:

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

Now we can get at the elements we want without using as many numbers. Instead of using `beatles[0][0]`, under our new structure now `beatles[0].name` is "John"

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

```
var beatles = {};
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 we've shown you have been very simple. All we'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 equal sign (`=`) to perform assignment. The operator for addition is the plus sign (`+`), the operator for subtraction is the minus sign (`-`), division uses the backslash (`/`), 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 JavaScript is weakly typed. The number will automatically be converted to a string:

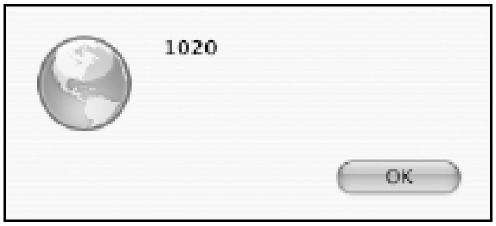
```
var year = 2005;
var message = "The year is " + year;
```

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);
```

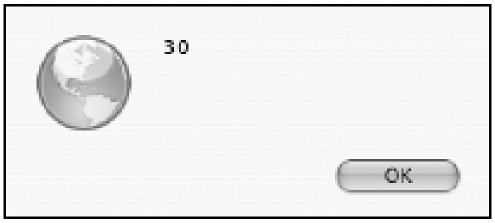
The first `alert` returns the string "1020." The second returns the number 30.

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



```
1020
```

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



```
30
```

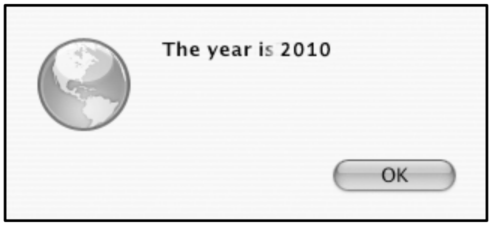
Another useful shorthand operator is += which performs addition and assignment (or concatenation and assignment) at the same time:

```
var year = 2010;
var message = "The year is ";
message += year;
```

The value of `message` is now “The year is 2010”. 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:



```
The year is 2010
```

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.

We've indented everything between the curly braces. This is not a syntax requirement of JavaScript—we've done it purely to make our 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:

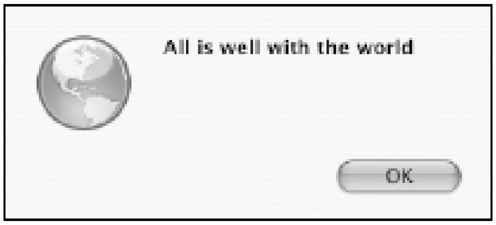
```
if (1 > 2) alert("The world has gone mad!");
```

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");
}
```

The following output appears when 1>2 is false:



All is well with the world

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 equal signs (`==`). Remember, a single equal sign is used for assignment. If you use a single equal 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.");
}
```

The problem is that we'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 we 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 equal sign (!=).

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

One confusing aspect of the == equality operator is that it's not strict. For example, what if we compare two values such as false and an empty string?

```
var a = false;
var b = "";
if (a == b) {
  alert("a equals b");
}
```

The result of this conditional statement evaluates to true, but why? The == equality operator compares false to an empty string and considers "an empty string" to have the same meaning as "false." For a strict comparison, you need to add another equal sign (===). This will perform a strict comparison of both the value *and* the type of the variable:

```
var a = false;
var b = "";
if (a === b) {
  alert("a equals b");
}
```

In this case the conditional statement evaluates to false because even though false and an empty string are considered the same, a Boolean and a String are not.

The same is true for the != inequality operator. For a strict comparison use !== instead.

Logical operators

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

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

This code uses 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, which 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 we've placed the operand in parentheses to avoid any ambiguities. We 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. The following statement uses 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. Its only drawback 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.

The while loop

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 this code. We began by creating a numeric variable, `count`, containing the value one. Then we 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.

The do...while loop

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.

The for loop

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 `while` loop we've already used. If we look at our `while` 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 `while` 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. It's important to remember that the index of the array begins at 0, not 1. In the following example, the array has four elements. The `count` variable increases from 0 once for every element in the array. When count reaches 4, the test condition fails and the loop ends, leaving 3 as the last index that was retrieved from the 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 reuse 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.

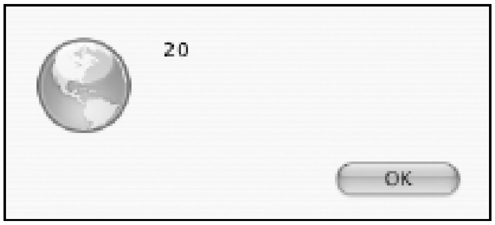
Here's a function that takes two arguments. If you pass this function two numbers, the function will multiply them:

```
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:



```
20
```

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;
}
```

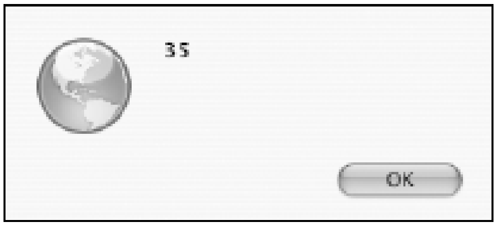
Here's a function that takes one argument (a temperature in degrees Fahrenheit) and returns a number (the same temperature in degrees Celsius):

```
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:


35

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 we've named my variables and functions. For variables, we've used underscores to separate words. For my functions, we've used capital letters after the first word (following the camel case naming style mentioned earlier). We've done this purely for that we 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

We'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.

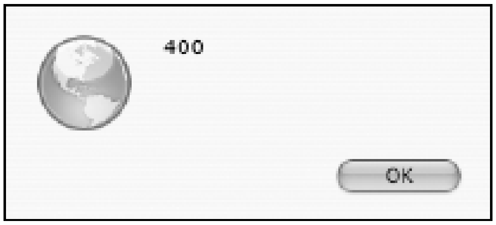
Fortunately, you can use the `var` keyword to explicitly set the scope of a variable within a function.

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:


400

The value of the variable `total` is now 400. All we wanted from the `square()` function was for it to return the value of `number` squared. But because we 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 we should have written the function:

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

Now we 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`. We could use the properties of the `Person` object to retrieve information about `jeremy`:

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

We'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 premade 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 prefilled 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 premade 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.

We're 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, we've shown you the basics of the JavaScript language. Throughout the rest of the book, we'll be using terms that have been introduced here: statements, variables, arrays, functions, and so on. 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.

We'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. We 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, we will introduce you to the idea of the DOM and show you how to use some of its very powerful methods.