CHAPTER 1

Getting Started with JavaScript

This book is about a scripting language called **JavaScript** and how to use it in a practical manner. After you read it, you'll be able to

- Understand JavaScript syntax and structures.
- Create scripts that are easy to understand and maintain.
- Write scripts that do not interfere with other JavaScripts.
- Write scripts that make web sites easier to use without blocking out non-JavaScript users.
- Write scripts that are independent of the browser or user agent trying to understand them—which means that in some years they will still be usable and won't rely on obsolete technology.
- Enhance a web site with JavaScript and allow developers without any scripting knowledge to change the look and feel.
- Enhance a web document with JavaScript and allow HTML developers to use your functionality by simply adding a CSS class to an element.
- Use progressive enhancement to make a web document nicer only when and if the user agent allows for it.
- Use Ajax to bridge the gap between back end and client side, thus creating sites that are easier to maintain and appear much slicker to the user.
- Use JavaScript as part of a web methodology that enables you to maintain it independently without interfering with the other development streams.

What you will not find here are

- Instructions on how to create effects that look flashy but do nothing of value for the visitor
- JavaScript applications that are browser specific
- JavaScripts that are only there to prove that they can be used and do not enhance the visitor's experience
- JavaScripts that promote unwanted content, such as pop-up windows or other flashy techniques like tickers or animation for animation's sake

It is my credo that JavaScript has a place in modern web development, but we cannot take it for granted that the visitor will be able to use or even experience all the effects and functionality we can achieve with JavaScript. JavaScript allows us to completely change the web page by adding and removing or showing and hiding elements. We can offer users richer interfaces like drag-and-drop applications or multilevel drop-down menus. However, some visitors cannot use a drag-and-drop interface because they can only use a keyboard or rely on voice recognition to use our sites. Other visitors might be dependent on *hearing* our sites rather than seeing them (via screen readers) and will not necessarily be notified of changes achieved via JavaScript. Last but not least, there are users who just cannot have JavaScript enabled, for example, in high-security environments like banks. Therefore, it is necessary to back up a lot of the things we do in JavaScript with solutions on the server side.

Sadly, JavaScript also has a history of being used as a way to force information onto the visitor that was not requested (pop-up windows are a good example). This practice is frowned on by me, as well as many professional web designers. It is my hope that you will not use the knowledge gained from this book to such an end.

■ **Note** Web design has matured over the years—we stopped using `FONT` tags and deprecated visual attributes like `bgcolor` and started moving all the formatting and presentational attributes to a CSS file. The same cleaning process has to happen to JavaScript should it remain a part of web development. We separated content, structure, and presentation, and now it is time to separate the behavior of web sites from the other layers. Web development now is for business and for helping the user rather than for the sake of putting something out there and hoping it works in most environments.

It is high time we see JavaScript as a part of an overall development methodology, which means that we develop it not to interfere with other technologies like HTML or CSS, but to interact with them or complement them. To that end, we see the emergence of a new technology (or at least a new way of using existing technologies) called **Ajax**, which we will discuss in Chapter 8.

Web development has come quite a way since the 1990s, and there is not much sense in creating web sites that are static and fixed in their size. Any modern web design should allow for growth as needed. It should also be accessible to everyone (which does not mean that everybody gets the same appearance—a nice multicolumn layout, for example, might make sense on a high-resolution monitor but is hard to use on a mobile phone or a PDA)—and ready for internationalization. We cannot afford any longer to build something and think it'll last forever. Since the Web is about content and change, it'll become obsolete if we don't upgrade our web products constantly and allow other data sources to feed into it or get information from it.

Enough introductions—you got this book to learn about JavaScript, so let's start by talking quickly about JavaScript's history and assets before diving right into it.

In this chapter you'll learn

- What JavaScript is and what it can do for you
- The advantages and disadvantages of JavaScript
- How to add JavaScript to a web document and its essential syntax
- Object-oriented programming (OOP) in relation to JavaScript
- How to write and run a simple JavaScript program

Chances are that you have already come across JavaScript, and already have an idea of what it is and what it can do, so we'll move quite swiftly through some basics of the language and its capabilities first. If you know JavaScript well already, and you simply want to know more about the newer and more accessible features and concepts, you might skip to Chapter 3. I won't hold it against you—however, there might be some information you've forgotten, and a bit of a review doesn't hurt, either.

The Why of JavaScript

In the beginning of the Web, there was HTML and the Common Gateway Interface (CGI). HTML defines the parts of a text document and instructs the user agent (usually the web browser) how to show it—for example, text surrounded by the tags `<p></p>` becomes a paragraph. Within that paragraph you may have `<h1></h1>` tags that define the main page heading. Notice that for most opening tags, there is a corresponding closing tag that begins with `</`.

HTML has one disadvantage—it has a fixed state. If you want to change something, or use data the visitor entered, you need to make a round-trip to a server. Using a **dynamic technology** (such as ColdFusion, ASP, ASP.NET, PHP, or JSP) you send the information from forms, or from parameters, to a server, which then performs calculating/testing/database lookups, etc. The application server associated with these technologies then writes an HTML document to show the results, and the resulting HTML document is returned to the browser for viewing.

The problem with that is it means every time there is a change, the entire process must be repeated (and the page reloaded). This is cumbersome, slow, and not as impressive as the new media "Internet" promised us to be. It is true that at least the Western world has the benefit of fast Internet connections these days, but displaying a page still means a reload, which could be a slow process that frequently fails (ever get an `Error 404`?).

We need something slicker—something that allows web developers to give immediate feedback to the user and change HTML without reloading the page from the server. Just imagine a form that needs to be reloaded every time there's an error in one of its fields—isn't it handier when something flags the errors immediately, without needing to reload the page from the web server? This is one example of what JavaScript can do for you.

Some information, such as calculations and verifying the information on a form, may not need to come from the server. JavaScript is executed by the user agent (normally a browser) on the visitor's computer. We call this **client-side code**. This could result in fewer trips to the server and faster-running web sites.

What Is JavaScript?

JavaScript started life as **LiveScript**, but Netscape changed the name—possibly because of the excitement being generated by Java—to JavaScript. The name is confusing though, as there is no real connection between Java and JavaScript—although some of the syntax looks similar.

Java is to JavaScript what Car is to Carpet

——From a JavaScript discussion group on Usenet

Netscape created the JavaScript language in 1996 and included it in their Netscape Navigator (NN) 2.0 browser via an interpreter that read and executed the JavaScript added to `.html` pages. The language has steadily grown in popularity since then, and is now supported by the most popular browsers.

The good news is that this means JavaScript can be used in web pages for all major modern browsers. The not-quite-so-good news is that there are differences in the way the different browsers implement JavaScript, although the core JavaScript language is much the same. However, JavaScript can be turned off by the user—and many companies and other institutions require their users to do so for security reasons. We will discuss this further shortly, as well as throughout this book.

The great thing about JavaScript is that once you've learned how to use it for browser programming, you can move on to use it in other areas. Microsoft's server—IIS—uses JavaScript to program server-side web pages (ASP), PDF files now use JavaScript, and even Windows administration tasks can be automated with JavaScript code. A lot of applications such as Dreamweaver and Photoshop are scriptable with JavaScript. Operating system add-ons like the Apple Dashboard or Konfabulator on Linux and Windows even allow you to write small helper applications in JavaScript.

Lately a lot of large companies also offer application programming interfaces (APIs) that feature JavaScript objects and methods you can use in your own pages—Google Maps being one of them. You can offer a zoomable and scrollable map in your web site with just a few lines of code.

Even better is the fact that JavaScript is a lot easier to develop than higher programming languages or server-side scripting languages. It does not need any compilation like Java or C++, or to be run on a server or command line like Perl, PHP, or Ruby: all you need to write, execute, debug, and apply JavaScript is a text editor and a browser—both of which are supplied with any operating system. There are, of course, tools that make it a lot easier for you, examples being JavaScript debuggers like Mozilla Venkman, Microsoft Script Debugger, or kjscmd.

Problems and Merits of JavaScript

As I mentioned at the outset of this chapter, JavaScript has been an integral part of web development over the last few years, but it has also been used wrongly. As a result, it has gotten a bad reputation. The reason for this is gratuitous JavaScript effects, like moving page elements and pop-up windows, which might have been impressive the first time you saw them but soon turned out to be just a "nice to have" and in some cases even a "nice to not have any longer." A lot of this comes from the days of **DHTML** (more on this in Chapter 3).

The term **user agent** and the lack of understanding what a user agent is can also be a problem. Normally, the user agent is a browser like Microsoft Internet Explorer (MSIE), Netscape, Mozilla (Moz), Firefox (Fx), Opera, or Safari. However, browsers are not the only user agents on the Web. Others include

- Assistive technology that helps users to overcome the limitations of a disability—like text-to-speech software or Braille displays
- Text-only agents like Lynx
- Web-enabled applications
- Game consoles
- Mobile/cell phones
- PDAs
- Interactive TV set-top boxes
- Search engines and other indexing programs
- And many more

This large variety of user agents, of different technical finesse (and old user agents that don't get updated), is also a great danger for JavaScript.

Not all visitors to your web site will experience the JavaScript enhancements you applied to it. A lot of them will also have JavaScript turned off—for security reasons. JavaScript can be used for good and for evil. If the operating system—like unpatched Windows—allows you to, you can install viruses or Trojan Horses on a computer via JavaScript or read out user information and send it to another server.

■ **Note** There is no way of knowing what the visitor uses or what his computer is capable of. Furthermore, you never know what the visitor's experience and ability is like. This is one of the beautiful aspects of the Web—everyone can participate. However, this can introduce a lot of unexpected consequences for the JavaScript programmer.

In many cases, you might want to have a server-side backup plan. It would test to see whether the user agent supports the functionality desired and, if it doesn't, the server takes over.

Independence of scripting languages is a legal requirement for web sites, defined in the Digital Discrimination Act for the UK, section 508 in the US law, and many more localized legal requirements throughout the world. This means that if the site you developed cannot be used without JavaScript, or your JavaScript enhancements are expecting a certain ability of the users or their user agent without a fallback, your client could be sued for discrimination.

However, JavaScript is not evil or useless, and it is a great tool to help your visitor to surf web sites that are a lot slicker and less time-consuming.

Why Use JavaScript If It Cannot Be Relied On?

As I just mentioned, just because it may not always be available doesn't mean that JavaScript shouldn't be used at all. It should simply not be the only means of user interaction.

The merits of using JavaScript are

- **Less server interaction**: You can validate user input before sending the page off to the server. This saves server traffic, which means saving money.
- **Immediate feedback to the visitors**: They don't have to wait for a page reload to see if they have forgotten to enter something
- **Automated fixing of minor errors**: For example, if you have a database system that expects a date in the format *dd-mm-yyyy* and the visitor enters it in the form *dd/mm/yyyy*, a clever JavaScript script could change this minor mistake prior to sending the form to the server. If that was the only mistake the visitor made, you can save her an error message—thus making it less frustrating to use the site.
- **Increased usability by allowing visitors to change and interact with the user interface without reloading the page**: For example, by collapsing and expanding sections of the page or offering extra options for visitors with JavaScript. A classic example of this would be select boxes that allow immediate filtering, such as only showing the available destinations for a certain airport, without making you reload the page and wait for the result.
- **Increased interactivity**: You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard. This is partly possible with CSS and HTML as well, but JavaScript offers you a lot wider—and more widely supported—range of options.
- **Richer interfaces**: If your users allow for it, you can use JavaScript to include such items as drag-and-drop components and sliders—something that originally was only possible in thick client applications your users had to install, such as Java applets or browser plug-ins like Flash.

- **Lightweight environment**: Instead of downloading a large file like a Java applet or a Flash movie, scripts are small in file size and get cached (held in memory) once they have been loaded. JavaScript also uses the browser controls for functionality rather than its own user interfaces like Flash or Java applets do. This makes it easier for users, as they already know these controls and how to use them. Modern Flash and Macromedia Flex applications do have the option to stream media and—being vector based—are visually scalable, something JavaScript and HTML controls aren't. On the other hand, they require the plug-in to be installed.

JavaScript in a Web Page and Essential Syntax

Applying JavaScript to a web document is very easy; all you need to do is to use the script tag:

```
<script type="text/javascript">
  // Your code here
</script>
```

For older browsers, or if you want to use strict XHTML (the newest version of HTML) instead of transitional, you'll need to comment out the code to make sure the user agent does not display it inside the page or tries to render it as HTML markup. There are two different syntaxes for commenting out code. For HTML documents and transitional XHTML, you use the normal HTML comments:

```
<script type="text/javascript">
<!--
 // Your code here
-->
</script>
```

In strict XHTML, you will need to use the CDATA commenting syntax to comment out your code—however, it is best not to add any JavaScript inside strict XHTML documents, but keep it in its own document. More on this in Chapter 3.

```
<script type="text/javascript"><!--//--><![CDATA[//><!–
  // Your code here
//--><!]]></script>
```

Technically it is possible to include JavaScript anywhere in the HTML document, and browsers will interpret it. However, there are reasons in modern scripting why this is a bad idea. For now though, we will add JavaScript examples to the body of the document to allow you to see immediately what your first scripts are doing. This will help you get familiar with JavaScript a lot easier than the more modern and advanced techniques awaiting you in Chapter 3.

■ **Note** There is also an "opposite" to the `script` tag—`noscript`—which allows you to add content that will only be displayed when JavaScript is not available. However, `noscript` is deprecated in XHTML and strict HTML, and there is no need for it—if you create JavaScript that is unobtrusive.

JavaScript Syntax

Before we go any further, we should discuss some JavaScript syntax essentials:

- `//` indicates that the rest of the current line is a comment and not code to be executed, so the interpreter doesn't try to run it. Comments are a handy way of putting notes in the code to remind us what the code is intended to do, or to help anyone else reading the code see what's going on.
- `/*` indicates the beginning of a comment that covers more than one line.
- `*/` indicates the end of a comment that covers more than one line. Multiline comments are also useful if you want to stop a certain section of code from being executed but don't want to delete it permanently. If you were having problems with a block of code, for example, and you weren't sure which lines were causing the problem, you could comment one portion of it at a time in order to isolate the problem.
- Curly braces ({ and }) are used to indicate a block of code. They ensure that all the lines inside the braces are treated as one block. You will see more of these when we discuss structures such as `if` or `for`, as well as functions.
- A semicolon or a newline defines the end of a statement, and a statement is a single command. Semicolons are in fact optional, but it's still a good idea to use them to make clear where statements end, because doing so makes your code easier to read and debug. (Although you can put many statements on one line, it's best to put them on separate lines in order to make the code easier to read.) You don't need to use semicolons after curly braces.

Let's put this syntax into a working block of code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
<head>
<body>
<script type="text/JavaScript">
  // One-line comments are useful for reminding us what the code is doing

  /*
     This is a multiline comment. It's useful for longer comments and
     also to block out segments of code when you're testing
  */

  /*
    Script starts here. We're declaring a variable myName, and assigning to it the
    value of whatever the user puts in the prompt box (more on that in Chapter
    2), finishing the instruction with a semicolon because it is a statement
  */
```

```
  var myName = prompt ("Enter your name","");

  // If the name the user enters is Chris Heilmann
  if (myName == "Chris Heilmann")
  {
     // then a new window pops up saying hello
     alert("Hello Me");
  }

  // If the name entered isn't Chris Heilmann
  else
  {
     // say hello to someone else
     alert("hello someone else");
  }
</script>
</body>
</html>
```

Some of the code may not make sense yet, depending on your previous JavaScript experience. All that matters for now is that it's clear how comments are used, what a code block is, and why there are semicolons at the end of some of the statements. You can run this script if you like—just copy it into an HTML page, save the document with the file extension `.html`, and open it in your browser.

Although statements like `if` and `else` span more than one line and contain other statements, they are considered single statements and don't need a semicolon after them. The JavaScript interpreter knows that the lines linked with an `if` statement should be treated as one block because of the curly braces, {}. While not mandatory, it is a good idea to indent the code within the curly braces. This makes reading and debugging much easier. We'll be looking at variables and conditional statements (`if` and `else`) in the next chapter.

Code Execution

The browser reads the page from top to bottom, so the order in which code executes depends on the order of the script blocks. A **script block** is the code between the `<script>` and `</script>` tags. (Also note that it's not just the browser that can read our code; the user of a web site can view your code, too, so it's not a good idea to put anything secret or sensitive in there.) There are three script blocks in this next example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
<head>
<script type="text/javascript">
alert( 'First script Block ');
alert( 'First script Block - Second Line ');
</script>
```

```
</head>
<body>
<h1>Test Page</h1>
<script type="text/JavaScript">
  alert( 'Second script Block' );
</script>
<p>Some more HTML</p>
<script type="text/JavaScript">
  alert( 'Third script Block' );
  function doSomething() {
    alert( 'Function in Third script Block' );
  }
</script>
</body>
</html>
```

If you try it out, you'll see that the `alert()` dialog in the first script block appears first displaying the message

```
First script Block
```

followed by the next `alert()` dialog in the second line displaying the message

```
First script Block - Second Line.
```

The interpreter continues down the page and comes to the second script block, where the `alert()` function displays this dialog:

```
Second script Block
```

and the third script block following it with an `alert()` statement that displays

```
Third script Block
```

Although there's another `alert` statement inside the function a few lines down, it doesn't execute and display the message. This is because it's inside a function definition (`function doSomething()`) and code inside a function executes only when the function is called.

An Aside About Functions

We'll be talking about functions in much more depth in Chapter 3, but I introduce them here because you can't get very far in JavaScript without an understanding of functions. A *function* is a named, reusable block of code, surrounded by curly braces, that you create to perform a task. JavaScript contains functions that are available for us to use and perform tasks like displaying a message to the user. Proper use of functions can save a programmer a lot of writing of repetitive code.

We can also create our own functions, which is what we did in the previous code block. Let's say we create some code that writes out a message to a page in a certain element. We'd probably want to use it again and again in different situations. While we could cut and paste code blocks wherever we wanted to use them, this approach can make the code excessively long; if you want the same piece of code three or four times within one page, it'll also get pretty hard to decipher and debug. Instead we can wrap the messaging code into a function and then pass in any information that the function needs in order to work using **parameters**. A function can also return a value to the code that called the function into action originally.

To call the function, you simply write its name followed by parentheses, `()`. (Note—you use the parentheses to pass the parameters. However, even when there are no parameters, you must still use the parentheses.) But you can't call the function, as you might expect, until the script has created it. We can call it in this script by adding it to the third script block like this:

```
<script type="text/JavaScript">
alert( 'Third script Block ');
function doSomething(){
  alert( 'Function in Third script Block ');
}
// Call the function doSomething
doSomething();
</script>
</body>
</html>
```

So far in this chapter you've looked at the pros and cons of the JavaScript language, seen some of the syntax rules, learned about some of the main components of the language (albeit briefly), and run a few JavaScript scripts. You've covered quite a lot of distance. Before we move on to a more detailed examination of the JavaScript language in the next chapter, let's talk about something key to successful JavaScript development: **objects**.

Objects

Objects are central to the way we use JavaScript. Objects in JavaScript are in many ways like objects in the world outside programming (it does exist, I just had a look). In the real world, an object is just a "thing" (many books about object-oriented programming compare objects to nouns): a car, a table, a chair, and the keyboard I'm typing on. Objects have

Properties (analogous to adjectives): The car is *red*.

Methods (like verbs in a sentence): The method for starting the car might be *turn ignition key*.

Events: Turning the ignition key results in the *car starting* event.

Object Oriented Programming (OOP) tries to make programming easier by modeling real-world objects. Let's say we were creating a car simulator. First, we would create a car object, giving it properties like *color* and *current speed*. Then we'd need to create methods: perhaps a *start* method to start the car, and a *break* method to slow the car, into which we'd need to pass information about how hard the brakes should be pressed so that we can determine the slowing effect. Finally, we would want some events, for example, a *gasoline low* event to remind us to fill up the car.

Object-oriented programming works with these concepts. This way of designing software is now very commonplace and influences many areas of programming—but most importantly to us, it's central to JavaScript and web browser programming.

Some of the objects we'll be using are part of the language specification: the `String` object, the `Date` object, and the `Math` object, for example. The same objects would be available to JavaScript in a PDF file and on a web server. These objects provide lots of useful functionality that could save us tons of programming time. The `Date` object, for example, allows you to obtain the current date and time from the client (such as a user's PC). It stores the date and provides lots of useful date-related functions, for example, converting the date/time from one time zone to another. These objects are usually referred to as **core objects**, as they are independent of the implementation. The browser also makes itself available for programming through objects that allow us to obtain information about the browser and to change the look and feel of the application. For example, the browser makes available the `Document` object, which represents a web page available to JavaScript. We can use this in JavaScript to add new HTML to the web page being viewed by the user of the web browser. If you were to use JavaScript with a different host, with a Windows server for example, then you'd find that the server hosting JavaScript exposes a very different set of host objects, their functionality being related to things you want to do on a web server.

You'll also see in Chapter 3 that JavaScript allows us to create our own objects. This is a powerful feature that allows us to model real-world problems using JavaScript. To create a new object, we need to specify the properties and methods it should have using a template called a **class**. A class is a bit like an architect's drawing in that it specifies what should go where and do what, but it doesn't actually create the object.

■ **Note** There is some debate as to whether JavaScript is an object-based language or an object-oriented language. The difference is that an object-based language uses objects for doing programming but doesn't allow the coder to use object-oriented programming in their code design. An object-oriented programming language not only uses objects, but also makes it easy to develop and design code in line with object-oriented design methodology. JavaScript allows us to create our own objects, but this is not accomplished in the same way as in class-based languages like Java or C#. However, we'll be concentrating not on debates about what is or isn't object oriented here, but on how objects are useful in practical terms in this book, and we'll look at some basic object-oriented coding where it helps make life easier for us.

As you progress through the book, you'll get a more in-depth look at objects: the objects central to the JavaScript language, the objects that the browser makes available for access and manipulation using JavaScript, and creating your own custom objects. For now, though, all you need to know is that objects in JavaScript are "entities" you can use to add functionality to web pages, and that they can have properties and methods. The `Math` object, for example, has among its properties one that represents the value of pi and among its methods one that generates a random number.

Simple JavaScript Example

We'll finish the chapter with a simple script that determines first the width of the visitor's screen and then applies a suitable style sheet (by adding an extra `LINK` element to the page). We'll do this using the `Screen` object, which is a representation of the user's screen. This object has an `availWidth` property that we'll retrieve and use to decide which style sheet to load.

Here's the code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    <title>CSS Resolution Demo</title>
    <!-- Basic style with all settings -->
    <link rel="StyleSheet" href="basic.css" type="text/css" />
    <!--
    Extra style (applied via JavaScript) to override default settings
    according to the screen resolution
    -->
```

```
    <script type="text/javascript">
      // Define a variable called cssName and a message
      // called resolutionInfo
      var cssName;
      var resolutionInfo;
      // If the width of the screen is less than 650 pixels
      if( screen.availWidth < 650 ) {
      // define the style Variable as the low-resolution style
        cssName = 'lowres.css';
        resolutionInfo = 'low resolution';
      // Or if the width of the screen is less than 1000 pixels
      } else {
        if( screen.availWidth > 1000 ) {
      // define the style Variable as the high-resolution style
          cssName = 'highres.css';
          resolutionInfo = 'high resolution';
      // Otherwise
        } else {
      // define the style Variable as the mid-resolution style
          cssName = 'lowres.css';
          resolutionInfo = 'medium resolution';
        }
      }
      document.write( '<link rel="StyleSheet" href="' +
      cssName + '" type="text/css" />' );
    </script>
    </head>
  <body>
    <script type="text/javascript">
      document.write( '<p>Applied Style:' +
      resolutionInfo + '</p>' );
    </script>
  </body>
</html>
```

Although we'll be looking at the details of `if` statements and loops in the next chapter, you can probably see how this is working already. The `if` statement on the first line asks whether the `screen.availWidth` is less than 650:

```
if ( screen.availWidth < 650 )
```

If the user's screen is 640×480, then the width is less than 650, so the code within the curly braces is executed and the low-resolution style and message get defined.

```
if ( screen.availWidth < 650 ) {
// define the style Variable as the low-resolution style
  cssName = 'lowres.css';
  resolutionInfo = 'low resolution';
}
```

The code carries on checking the screen size using the `else` statement. The final `else` only occurs if neither of the other evaluations have resulted in code being executed, so we assume that the screen is 800× 600, and define the medium style and message accordingly:

```
else {
// define the style Variable as the mid-resolution style
  cssName = 'lowres.css';
  resolutionInfo = 'medium resolution';
}
```

It's also worth noting that we're measuring the screen size here, and the user may have a 800× 600 screen, but that doesn't mean their browser window is maximized. We may be applying a style that may not be appropriate.

We're using another object, the `document` object, to `write` to the page (HTML document). The `document` object's `write()` method allows us to insert HTML into the page. Note that `document.write()` doesn't actually change the source HTML page, just the page the user sees on his computer.

■ **Note** In fact, you'll find `document.write()` very useful as you work through the first few chapters of the book. It's good for small examples that show how a script is working, for communicating with the user, and even for debugging an area of a program that you're not sure is doing what you think it should be doing. It also works on all browsers that support JavaScript. More modern browsers have better tools and methods for debugging, but more on that in Chapter 3.

We use `document.write()` to write out the appropriate link element with our defined style in the head:

```
document.write( '<link rel="StyleSheet" href="' +
cssName + '" type="text/css" />' );
```

And in the document's body, we write out the message explaining which resolution style was applied:

```
<script type="text/javascript">
  document.write( '<p>Applied Style: '+ resolutionInfo + '</p>' );
</script>
```

Later on, we'll be working with more complex examples that use JavaScript to test capabilities of the user's agent and interface. For now though, I hope this simple example gives you an inkling of the kind of flexibility you can add to your web pages using JavaScript.

Summary

In this chapter, we've taken a look at what JavaScript is, how it works, and what its advantages and disadvantages are. I noted that the biggest disadvantage is that we cannot rely on it as a given. However, I also mentioned that using JavaScript can make web sites a nicer and slicker experience.

You've run some JavaScript code, seen how to add comments to the code, and how to separate JavaScript statements using semicolons. You also saw that you can tell JavaScript to treat a group of lines of code as a single block using curly braces, following an `if` statement, for example. You learned that JavaScript execution generally runs from top to bottom, and from the first `script` block to the last, with the exception of functions that only execute when you tell them to.

We also looked at objects, which are central to writing JavaScript. Not only is JavaScript itself very much dependent on objects, but the browser also uses objects and methods to make itself and the document available for scripting. Finally, we looked at a simple example that reads out the user's screen resolution and applies a suitable style sheet.

In the next chapter, I'll cover the language fundamentals of JavaScript. You'll see how JavaScript stores and manipulates data, and uses it in calculations. We'll also look at creating "intelligent" JavaScript programs using decision-making statements that allow us to evaluate data, do calculations with it, and decide on an appropriate course of action. With that chapter under your belt, you'll have most of the fundamental knowledge needed to go on to more exciting and useful web programming.