

1

Introduction to JavaScript and the Web

In this introductory chapter, we'll take a look at what JavaScript is, what it can do for you, and what you need to be able to use it. With these foundations in place, we will see throughout the rest of the book how JavaScript can help you to create powerful web applications for your website.

The easiest way to learn something is by actually doing it, so throughout the book we'll be creating a number of useful example programs using JavaScript. We start this process in this chapter, by the end of which you will have created your first piece of JavaScript code.

Additionally over the course of the book, we'll develop a complete JavaScript web application: an online trivia quiz. By seeing it develop, step-by-step, you'll get a good understanding of how to create your own web applications. At the end of this chapter, we'll look at the finished trivia quiz and discuss the ideas behind its design.

Introduction to JavaScript

In this section, we're going to take a brief look at what JavaScript is, where it came from, how it works, and what sorts of useful things we can do with it.

What Is JavaScript?

Having bought this book you are probably already well aware that JavaScript is some sort of *computer language*, but what is a computer language? Put simply, a computer language is a series of instructions that instruct the computer to do something. That something can be a wide variety of things, including displaying text, moving an image, or asking the user for information. Normally the instructions, or what is termed *code*, are *processed* from the top line downward. Processed simply means that the computer looks at the code we've written, works out what action we want taken, and then takes that action. The actual act of processing the code is called *running* or *executing* it.

Chapter 1

Using natural English, let's see what instructions, or code, we might write to make a cup of coffee.

1. Put coffee in cup.
2. Fill kettle with water.
3. Put kettle on to boil.
4. Has the kettle boiled? If so, then pour water into cup; otherwise, continue to wait.
5. Drink coffee.

We'd start running this code from the first line (instruction 1), and then continue to the next (instruction 2), then the next, and so on until we came to the end. This is pretty much how most computer languages work, JavaScript included. However, there are occasions when we might change the flow of execution, or even skip over some code, but we'll see more of this in Chapter 3.

JavaScript is an interpreted language, rather than a compiled language. What do we mean by the terms interpreted and compiled?

Well, to let you in on a secret, your computer doesn't really understand JavaScript at all. It needs something to interpret the JavaScript code and convert it into something that it understands; hence it is an *interpreted language*. Computers only understand machine code, which is essentially a string of binary numbers (that is, a string of zeros and ones). As the browser goes through the JavaScript, it passes it to a special program called an *interpreter*, which converts the JavaScript to the machine code your computer understands. It's a bit like having a translator to translate English into Spanish, for example. The important point to note is that the conversion of the JavaScript happens at the time the code is run; it has to be repeated every time the code is run. JavaScript is not the only interpreted language; there are others, including VBScript.

The alternative *compiled language* is one where the program code is converted to machine code before it's actually run, and this conversion only has to be done once. The programmer uses a compiler to convert the code that he wrote to machine code, and it is this machine code that is run by the program's user. Compiled languages include Visual Basic and C++. Using a real-world analogy, it's like having someone translate our English document into Spanish. Unless we change the document, we can use it without retranslation as much as we like.

Perhaps this is a good point to dispel a widespread myth: JavaScript is not the script version of the Java language. In fact, although they share the same name, that's virtually all they do share. Particularly good news is that JavaScript is much, much easier to learn and use than Java. In fact, languages like JavaScript are the easiest of all languages to learn but are still surprisingly powerful.

JavaScript and the Web

For most of this book we'll be looking at JavaScript code that runs inside a web page loaded into a browser. All we need to create these web pages is a text editor, for example, Windows NotePad, and a web browser, such as Netscape Navigator or Internet Explorer, with which we can view our pages. These browsers come equipped with JavaScript interpreters.

In fact, the JavaScript language first became available in the web browser Netscape Navigator 2. Initially, it was called LiveScript. However, because Java was the hot technology of the time, Netscape decided

that JavaScript sounded more exciting. Once JavaScript really took off, Microsoft decided to add their own brand of JavaScript to Internet Explorer, which they named JScript. Since then, both Netscape and Microsoft have released improved versions and included them in their latest browsers. Although these different brands and versions of JavaScript have much in common, there are enough differences to cause problems if we're not careful. Initially we'll be creating code that'll work with Netscape and Microsoft version 4 and later browsers. Later chapters of the book look at features available only to Netscape 6+ and IE 5.5+. We'll look into the problems with different browsers and versions of JavaScript later in this chapter, and see how we deal with them.

The majority of the web pages containing JavaScript that we will create in this book can be stored on your hard drive and loaded directly into your browser from the hard drive itself, just as you'd load any normal file (such as a text file). However, this is not how web pages are loaded when we browse websites on the Internet. The Internet is really just one great big network connecting computers together. Websites are a special service provided by particular computers on the Internet; the computers providing this service are known as *web servers*.

Basically the job of a web server is to hold lots of web pages on its hard drive. When a browser, usually on a different computer, requests a web page that is contained on that web server, the web server loads it from its own hard drive and then passes the page back to the requesting computer via a special communications protocol called *HyperText Transfer Protocol (HTTP)*. The computer running the web browser that makes the request is known as the client. Think of the client/server relationship as a bit like a customer/shopkeeper relationship. The customer goes into a shop and says, "Give me one of those." The shopkeeper serves the customer by reaching for the item requested and passing it back to the customer. In a web situation, the client machine running the web browser is like the customer and the web server getting the page requested is like the shopkeeper.

When we type an address into the web browser, how does it know which web server to get the page from? Well, just as shops have addresses, say, 45 Central Avenue, SomeTownsville, so do web servers. Web servers don't have street names; instead they have *Internet Protocol (IP) addresses*, which uniquely identify them on the Internet. These consist of four sets of numbers, separated by dots; for example, 127.0.0.1.

If you've ever surfed the net, you're probably wondering what on earth I'm talking about. Surely web servers have nice `www.somewebsite.com` names, not IP addresses? In fact, the `www.somewebsite.com` name is the "friendly" name for the actual IP address; it's a whole lot easier for us humans to remember. On the Internet, the friendly name is converted to the actual IP address by computers called *domain name servers*, something your Internet service provider will have set up for you.

Toward the end of the book, we'll go through the process of how to set up our own web server in a step-by-step guide. We'll see that web servers are not just dumb machines that pass pages back to clients, but in fact they can do a bit of processing themselves using JavaScript. We'll be looking at this later in the book as well.

Why Choose JavaScript?

JavaScript is not the only scripting language; there are others such as VBScript and Perl. So why choose JavaScript over the others?

The main reason for choosing JavaScript is its widespread use and availability. Both of the most commonly used browsers, Internet Explorer and Netscape Navigator, support JavaScript, as do some of the

less commonly used browsers. So, basically we can assume that most people browsing our website will have a version of JavaScript installed, though it is possible to use a browser's options to disable it.

Of the other scripting languages we mentioned, VBScript, which can be used for the same purposes as JavaScript, is only supported by Internet Explorer running on the Windows operating system, and Perl is not used at all in web browsers.

JavaScript is also very versatile and not just limited to use within a web page. For example, it can be used in Windows to automate computer administration tasks and inside Adobe Acrobat.pdf files to control the display of the page just as in web pages, although this is a more limited version of JavaScript. However, the question of which scripting language is the most powerful and useful has no real answer. Pretty much everything that can be accomplished in JavaScript can be done in VBScript, and vice versa.

What Can JavaScript Do for Me?

The most common uses of JavaScript are interacting with users, getting information from them, and validating their actions. For example, say we want to put a drop-down menu on the page so that users can choose where they want to go to on our website. The drop-down menu might be plain old HTML, but it needs JavaScript behind it to actually do something with the user's input. Other examples of using JavaScript for interactions are given by forms, which are used for getting information from the user. Again these may be plain HTML, but we might want to check the validity of the information that the user is entering. For example, if we had a form taking a user's credit card details in preparation for the online purchase of goods, we'd want to make sure they had actually filled in their credit card details before we sent them the goods. We might also want to check that the data being entered is of the correct type, such as a number for their age rather than text.

JavaScript can also be used for various "tricks." One example is switching an image in a page for a different one when the user rolls her mouse over it, something often seen in web page menus. Also, if you've ever seen scrolling messages in the browser's status bar (usually at the bottom of the browser window) or inside the page itself and wondered how they manage that, this is another JavaScript trick that we'll demonstrate later in the book. We'll also see how to create expanding menus that display a list of choices when a user rolls his or her mouse over them, another commonly seen JavaScript-driven trick.

Tricks are OK up to a point, but even more useful are small applications that provide a real service. For example, a mortgage seller's website that has a JavaScript-driven mortgage calculator, or a website about financial planning that includes a calculator that works out your tax bill for you. With a little inventiveness you'll be amazed at what can be achieved.

Tools Needed to Create JavaScript Web Applications

All that you need to get started with creating JavaScript code for web applications is a simple text editor, such as Windows NotePad, or one of the many slightly more advanced text editors that provide line numbering, search and replace, and so on. An alternative is a proper HTML editor; you'll need one that allows you to edit the HTML source code, because that's where you need to add your JavaScript. A number of very good tools specifically aimed at developing web-based applications, such as Macromedia's

excellent Dreamweaver MX, are also available. However, in this book we'll be concentrating on JavaScript, rather than any specific development tool. When it comes to learning the basics, it's often best to write the code by hand rather than relying on a tool to do it for you. This helps you to understand the fundamentals of the language before you attempt the more advanced logic that is beyond a tool's capability. Once you have a good understanding of the basics, you can use tools as timesavers so that you can spend more time on the more advanced and more interesting coding.

You'll also need a browser to view your web pages in. It's best to develop your JavaScript code on the sort of browsers you expect visitors to use to access your website. We'll see later in the chapter that there are different versions of JavaScript, each supported by different versions of the web browsers. Each of these JavaScript versions, while having a common core, also contains various extensions to the language. All the examples that we give in this book have been tested on Netscape Navigator versions 4.0, 4.7, 6, and 7, and Internet Explorer versions 4.0 to 6.0. Wherever a piece of code does not work on any of these browsers, a note to this effect has been made in the text. (In case you're wondering, Netscape Navigator 5 never made it out on general release to the public.)

Even if your browser supports JavaScript, it is possible to disable this functionality in the browser. So, before we start on our first JavaScript examples in the next section, you should check whether JavaScript is enabled in your browser.

To do this in Netscape Navigator, choose Preferences from the Edit menu on the browser. In the window that appears, click the Advanced tab or item from the list. In Netscape 7 you need to click on Scripts and plug-ins, and check that the checkbox beside Enable JavaScript for Navigator is checked. If not, then check it. You can also change what JavaScript code is permitted to do; for example, you can modify the code to permit JavaScript to open new browser windows.

It is harder to turn off scripting in Internet Explorer. Choose Internet Options from the Tools menu on the browser, click the Security tab, and check whether the Internet or Local intranet options have custom security settings. If either of them do, click the Custom Level button, and scroll down to the Scripting section. Check that Active Scripting is set to Enable.

A final point to note is how to open our code examples in your browser. For most of the book, you simply need to open the file from where it is stored on your hard drive. You can do this in a number of ways. One way in Internet Explorer is to choose Open from the File menu, and click the Browse button to browse to where you stored the code. Similarly, in Netscape Navigator choose Open Page from the File menu, and click the Choose File button; in Netscape Navigator 6, choose Open File from the File menu.

The `<script>` Tag and Your First Simple JavaScript Program

We've discussed the subject of JavaScript for long enough; now let's look at how we put it into our web page. In this section, we'll write our first piece of JavaScript code.

Inserting JavaScript in a web page is much like inserting any other HTML content; we use tags to mark the start and end of our script code. The tag we use to do this is the `<script>` tag. This tells the browser that the following chunk of text, bounded by the closing `</script>` tag, is not HTML to be displayed,

Chapter 1

but rather script code to be processed. We call the chunk of code surrounded by the `<script>` and `</script>` tags a *script block*.

Basically when the browser spots `<script>` tags, instead of trying to display the contained text to the user, it uses the browser's built-in JavaScript interpreter to run the code's instructions. Of course, the code might give instructions about changes to the way the page is displayed or what is shown in the page, but the text of the code itself is never shown to the user.

We can put the `<script>` tags inside the header (between the `<head>` and `</head>` tags), or inside the body (between the `<body>` and `</body>` tags) of the HTML page. However, although we can put them outside these areas, for example, before the `<html>` tag or after the `</html>` tag, this is not permitted in the web standards and so is considered bad practice.

The `<script>` tag has a number of attributes, but the most important one for us are the `language` and `type` attributes. As we saw earlier, JavaScript is not the only scripting language available, and different scripting languages need to be processed in different ways. We need to tell the browser which scripting language to expect so that it knows how to process it. There are no prizes for guessing that the `language` attribute, when using JavaScript, takes the value `JavaScript`. So, our opening script tag will look like this:

```
<script language="JavaScript" type="text/javascript">
```

Including the `language` attribute is good practice, but within a web page it can be left off. Browsers such as Internet Explorer (IE) and Netscape Navigator (NN) default to a script language of JavaScript. This means that if the browser encounters a `<script>` tag with no `language` attribute set, it assumes that the script block is written in JavaScript. However, it is good practice to always include the `language` attribute.

In some situations JavaScript is not the default language, such as when script is run server-side (see Chapter 16), so we need to specify the language and sometimes the version of JavaScript that our web page requires. However, when not specifying the `language` attribute will cause problems, I'll be sure to warn you.

However, the web standards have depreciated the requirements for the `language` attribute; this means the final aim is for its removal. In its place we have the `type` attribute, and this is made mandatory by the web standards, although most current browsers will let us get away without including it. In the case of script, we need to specify that the type is `text/JavaScript`, as follows:

```
<script type="text/javascript">
```

However, older browsers do not support the `type` attribute, so for compatibility it's best to include both, as shown here:

```
<script language="JavaScript" type="text/javascript">
```

Also note that we can't currently specify the JavaScript version with the `type` attribute, only with the `language` attribute.

OK, let's take a look at our first page containing JavaScript code.

Try It Out Painting the Document Red

We'll try out a simple example of using JavaScript to change the background color of the browser. In your text editor (I'm using Windows NotePad), type in the following:

```
<html>
<body BGCOLOR="WHITE">
<p>Paragraph 1</p>
<script language="JavaScript" type="text/javascript">
    document.bgColor = "RED";
</script>
</body>
</html>
```

Save the page as `ch1_examp1.htm` to a convenient place on your hard drive. Now load it into your web browser. You should see a red web page with the text Paragraph 1 in the top left-hand corner. But wait—didn't we set the `<body>` tag's `BGCOLOR` attribute to white? OK, let's look at what's going on here.

How It Works

The page is contained within `<html>` and `</html>` tags. This then contains a `<body>` element. When we define the opening `<body>` tag, we use HTML to set the page's background color to white.

```
<BODY BGCOLOR="WHITE">
```

Then, we let the browser know that our next lines of code are JavaScript code by using the `<script>` start tag.

```
<script language="JavaScript" type="text/javascript">
```

Everything from here until the close tag, `</script>`, is JavaScript and is treated as such by the browser. Within this script block, we use JavaScript to set the document's background color to red.

```
document.bgColor = "RED";
```

What we might call the *page* is known as the *document* when scripting in a web page. The document has lots of properties, including its background color, `bgColor`. We can reference properties of the `document` by writing `document`, then putting a dot, then the property name. Don't worry about the use of the `document` at the moment; we'll be looking at it in depth later in the book.

Note that the preceding line of code is an example of a JavaScript *statement*. Every line of code between the `<script>` and `</script>` tags is called a statement, although some statements may run on to more than one line.

You'll also see that there's a semicolon (`;`) at the end of the line. We use a semicolon in JavaScript to indicate the end of a statement. In practice, JavaScript is very relaxed about the need for semicolons, and when you start a new line, JavaScript will usually be able to work out whether you mean to start a new line of code. However, for good coding practice, you should use a semicolon at the end of statements of code, and a single JavaScript statement should fit onto one line and shouldn't be continued onto two or more lines. Moreover, you'll find there are times when you must include a semicolon, which we'll come to later in the book.

Chapter 1

Finally, to tell the browser to stop interpreting our text as JavaScript and start interpreting it as HTML, we use the script close tag:

```
</script>
```

We've now looked at how the code works, but we haven't looked at the order in which it works. When the browser loads in the web page, the browser goes through it, rendering it tag by tag from top to bottom of the page. This process is called *parsing*. The web browser starts at the top of the page and works its way down to the bottom of the page. The browser comes to the `<body>` tag first and sets the document's background to white. Then it continues parsing the page. When it comes to the JavaScript code, it is instructed to change the document's background to red.

Try It Out The Way Things Flow

Let's extend the previous example to demonstrate the parsing of a web page in action. Type the following into your text editor:

```
<html>
<body BGCOLOR="WHITE">
<p>Paragraph 1</p>
<script language="JavaScript" type="text/javascript">
    // Script block 1
    alert("First Script Block");
</script>
<p>Paragraph 2</p>
<script language="JavaScript" type="text/javascript">
    // Script block 2
    document.bgColor = "RED";
    alert("Second Script Block");
</script>
<p>Paragraph 3</p>
</body>
</html>
```

Save the file to your hard drive as `ch1_examp2.htm`, and then load it into your browser. When you load the page you should see the first paragraph, Paragraph 1, appear followed by a message box displayed by the first script block. The browser halts its parsing until you click the OK button. As you can see in Figure 1-1, the page background is white, as set in the `<body>` tag, and only the first paragraph is currently displayed.

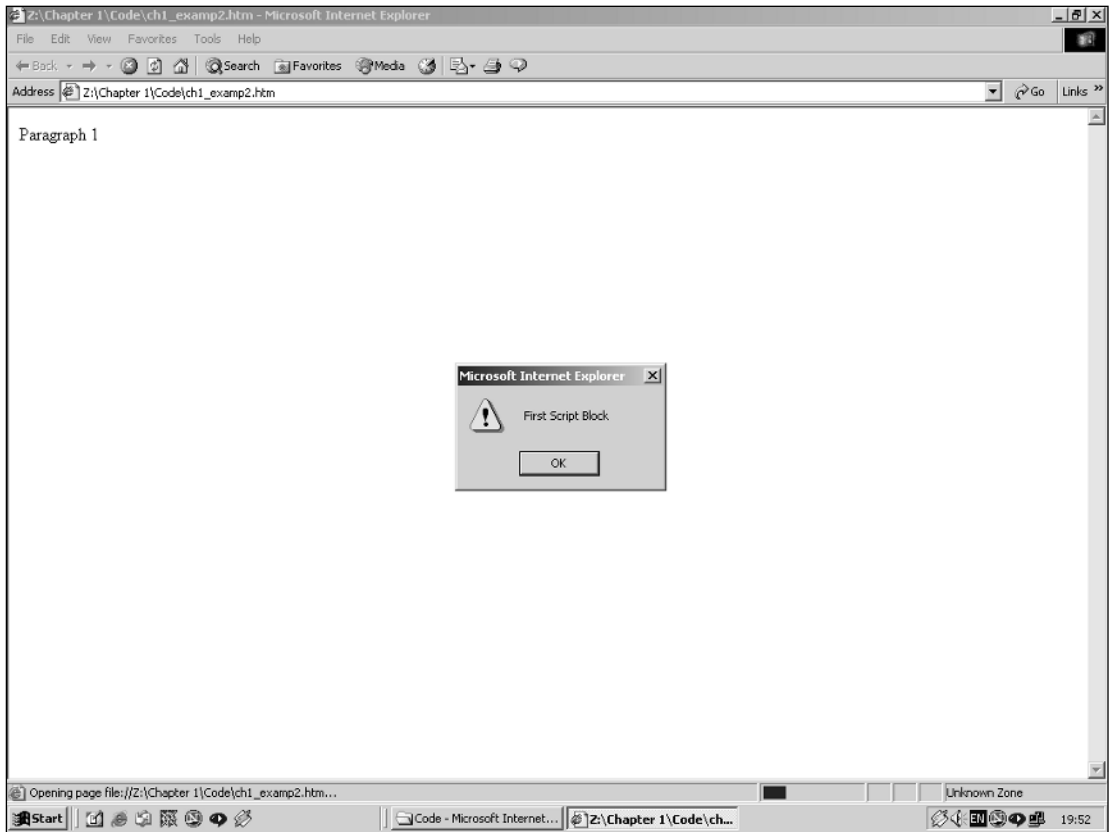


Figure 1-1

Click the OK button, and the parsing continues. The browser displays the second paragraph, and the second script block is reached, which changes the background color to red. Another message box is displayed by the second script block, as shown in Figure 1-2.

Chapter 1

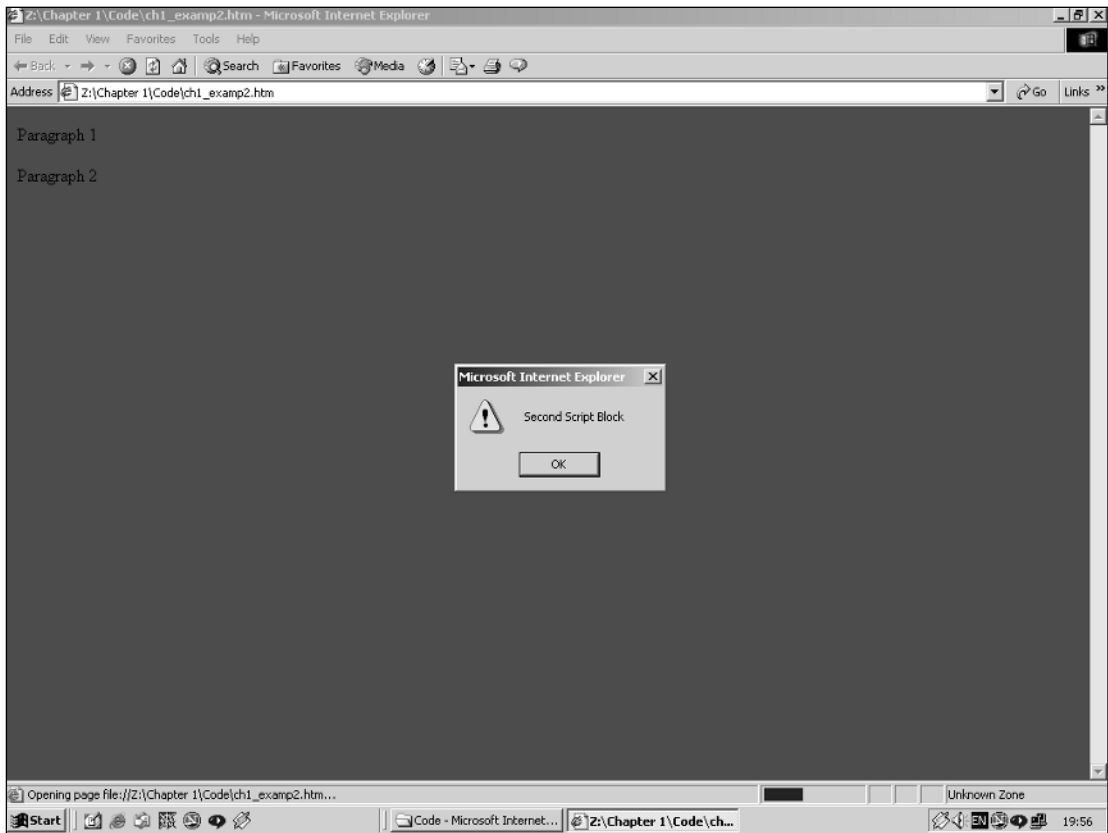


Figure 1-2

Click OK, and again the parsing continues, with the third paragraph, Paragraph 3, being displayed. The web page is complete, as shown in Figure 1-3.

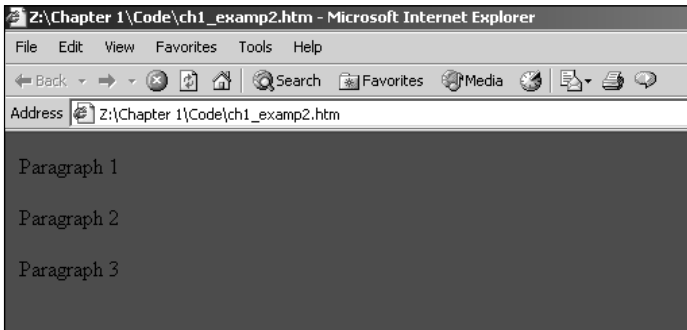


Figure 1-3

How It Works

The first part of the page is the same as in our earlier example. The background color for the page is set to white in the definition of the `<body>` tag, and then a paragraph is written to the page.

```
<html>
<body BGCOLOR="WHITE">
<p>Paragraph 1</p>
```

The first new section is contained in the first script block.

```
<script language="JavaScript" type="text/javascript">
  // Script block 1
  alert("First Script Block");
</script>
```

This script block contains two lines, both of which are new to us. The first line

```
// Script block 1
```

is just a *comment*, solely for our benefit. The browser recognizes anything on a line after a double forward slash (`//`) to be a comment and does not do anything with it. It is useful for us as programmers, because we can add explanations to our code that make it easier for us to remember what we were doing when we come back to our code at a later date.

The `alert()` function in the second line of code is also new to us. Before we can explain what it does, we need to explain what a *function* is.

We will define functions more fully in Chapter 3, but for now we just need to think of them as pieces of JavaScript code that we can use to do certain tasks. If you have a background in math, you may already have some idea of what a function is: A function takes some information, processes it, and gives you a result. A function makes life easier for us as programmers because we don't have to think about how the function does the task but can just concentrate on when we want the task done.

In particular, the `alert()` function enables us to alert or inform the user about something, by displaying a message box. The message to be given in the message box is specified inside the parentheses of the `alert()` function and is known as the function's *parameter*.

The message box that's displayed by the `alert()` function is *modal*. This is an important concept, which we'll come across again. It simply means that the message box won't go away until the user closes it by clicking the OK button. In fact, parsing of the page stops at the line where the `alert()` function is used and doesn't restart until the user closes the message box. This is quite useful for this example, because it allows us to demonstrate the results of what has been parsed so far: The page color has been set to white, and the first paragraph has been displayed.

Once you click OK, the browser carries on parsing down the page through the following lines:

```
<p>Paragraph 2</p>
<script language="JavaScript" type="text/javascript">
  // Script block 2
  document.bgColor = "RED";
  alert("Second Script Block");
</script>
```

The second paragraph is displayed, and the second block of JavaScript is run. The first line of the script block code is another comment, so the browser ignores this. We saw the second line of the script code in our previous example—it changes the background color of the page to red. The third line of code is our `alert()` function, which displays the second message box. Parsing is brought to a halt until we close the message box by clicking OK.

When we close the message box, the browser moves on to the next lines of code in the page, displaying the third paragraph and finally ending our web page.

```
<p>Paragraph 3</p>
</body>
</html>
```

Another important point raised by this example is the difference between setting properties of the page, such as background color, via HTML and doing the same thing using JavaScript. The method of setting properties using HTML is *static*: A value can be set only once and never changed again using HTML. Setting properties using JavaScript enables us to dynamically change their values. By *dynamic*, I simply mean something that can be changed and whose value or appearance is not set in stone.

Our example is just that, an example. In practice if we wanted the page's background to be red, we would set the `<body>` tag's `BGColor` attribute to "RED", and not use JavaScript at all. Where we would want to use JavaScript is where we want to add some sort of intelligence or logic to the page. For example, if the user's screen resolution is particularly low, we might want to change what's displayed on the page; with JavaScript, we can do this. Another reason for using JavaScript to change properties might be for special effects, for example, making a page fade in from white to its final color.

A Brief Look at Browsers and Compatibility Problems

We've seen in the preceding example that by using JavaScript we can change a web page's document's background color using the `bgColor` property of the document. The example worked whether you used a Netscape or Microsoft browser, and the reason for this is that both browsers support a document with a `bgColor` property. We can say that the example is *cross-browser compatible*. However, it's not always the case that the property or language feature available in one browser will be available in another browser. This is even sometimes the case between versions of the same browser.

The version numbers for Internet Explorer and Netscape Navigator browsers are usually written as a decimal number, for example Netscape Navigator has a version 4.06. In this book we will use the following terminology to refer to these versions. By version 4.x we mean all versions starting with the number 4. By version 4.0+ we mean all versions with a number greater than or equal to 4.

One of the main headaches involved in creating web-based JavaScript is the differences between different web browsers, the level of HTML they support, and the functionality their JavaScript interpreters can handle. You'll find that in one browser, you can move an image using just a couple of lines of code, and in another, it'll take a whole page of code, or even prove impossible. One version of JavaScript will contain a method to change text to uppercase, and another won't. Each new release of Microsoft or Netscape browsers sees new and exciting features added to their HTML and JavaScript support. The good news is

that with a little ingenuity we can write JavaScript that will work with both Microsoft and Netscape browsers.

Which browsers you want to support really comes down to the browsers you think the majority of your website's visitors, that is, your *user base*, will be using. This book has been aimed at both Internet Explorer 4 and above (IE 4.0+) and Netscape Navigator 4 and above (NN 4.0+). However, in later chapters we'll be concentrating on more modern browsers like IE6 and Netscape 7.

If we want our website to be professional, we need to somehow deal with older browsers. We could make sure our code is backward compatible—that is, it only uses features that were available in older browsers. However, we may decide that it's simply not worth limiting ourselves to the features of older browsers. In this case we need to make sure our pages degrade gracefully. In other words, although our pages won't work in older browsers, they will fail in a way that means the user is either never aware of the failure or is alerted to the fact that certain features on the website are not compatible with his or her browser. The alternative to degrading gracefully is for our code to raise lots of error messages, cause strange results to be displayed on the page, and generally make us look like idiots who don't know what we're doing!

So how do we make our web pages degrade gracefully? You can do this by using JavaScript to check which browser the web page is running in after it has been partially or completely loaded. We can use this information to determine what scripts to run or even to redirect the user to another page written to make best use of their particular browser. In later chapters, we'll see how to check for the browser version and take appropriate action so that our pages work acceptably on as many browsers as possible.

Following is a table listing the different versions of JavaScript (and JScript) that Microsoft and Netscape browsers support. However, it is a necessary oversimplification to some extent, because there is no exact feature-by-feature compatibility. We can only indicate the extent to which different versions have similarities. Also, as we'll see in Chapter 12, it's not just the JavaScript support that is a problem, but also the extent to which the HTML can be altered by code.

On a more positive note, the core of the JavaScript language does not vary too much between JavaScript versions; the differences are mostly useful extra features, which are nice-to-haves but often not essential. We'll concentrate on the core parts of the JavaScript language in the next few chapters.

Language Version	Netscape Navigator Version	Internet Explorer Version
JavaScript 1.0 (equivalent to JScript 1.0)	2.x	3.x
JavaScript 1.1	3.x	N/A
JavaScript 1.2 (equivalent to JScript 3.0)	4.0–4.05	4.x
JavaScript 1.3	4.06+	N/A
JavaScript 1.4 (equivalent to JScript 5.0)		5.0
JavaScript 1.5	6.x and 7.x	5.5 and 6.0

Introducing the Trivia Quiz

Over the course of the book, we'll be developing a full web-based application, namely a trivia quiz. The trivia quiz works with both Netscape Navigator 4.0+ and Internet Explorer 4.0+ web browsers, making full use of their JavaScript capabilities. Initially, the quiz runs purely using JavaScript code in the web browser, but later it will also use JavaScript running on a web server to access a Microsoft Access database containing the questions.

Let's take a look at what the quiz will finally look like. The main starting screen is shown in Figure 1-4. Here the user can choose the number of questions that they want to answer and whether to set themselves a time limit. Using a JavaScript-based timer, we keep track of when the time that they have allotted themselves is up.

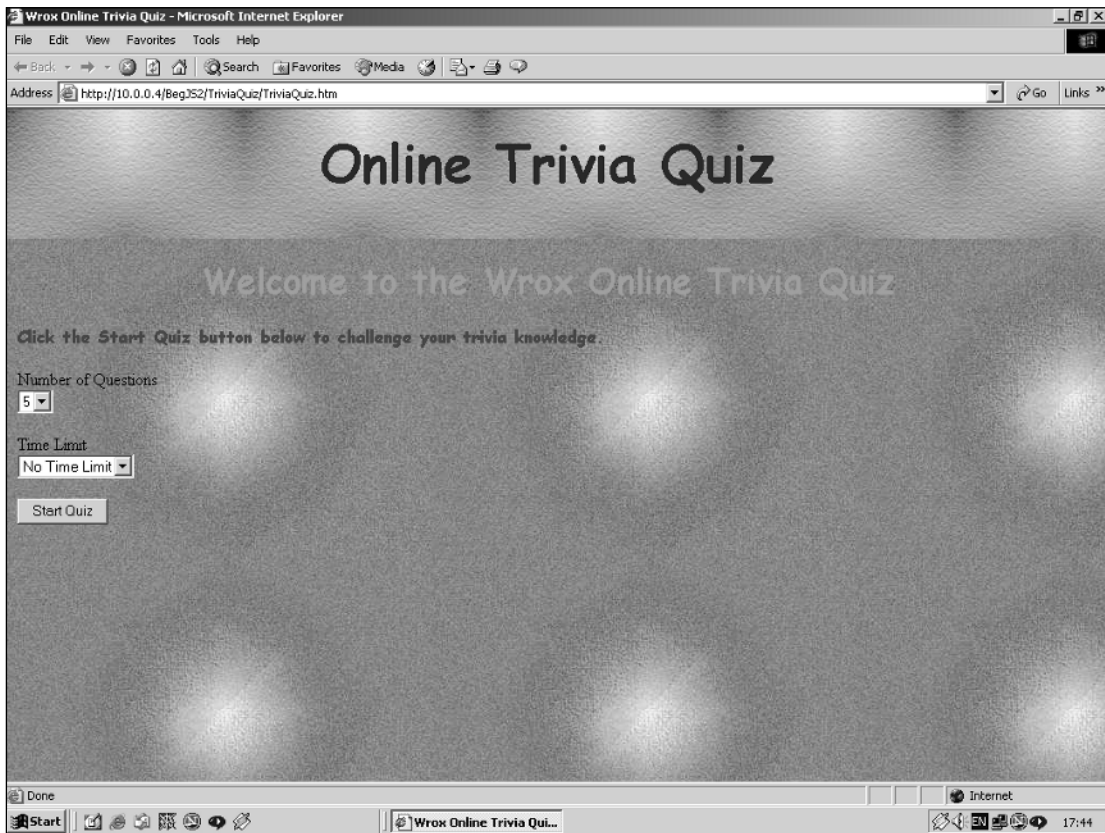
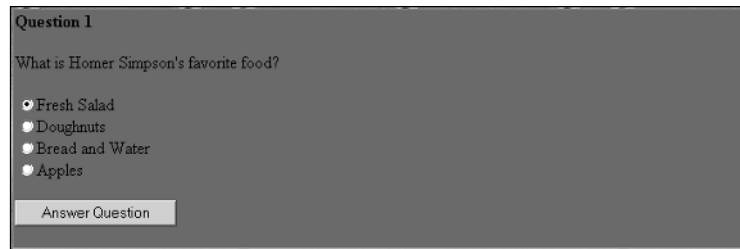


Figure 1-4

After clicking the Start Quiz button, the user is faced with a random choice of questions pulled from a database that we'll create to hold the trivia questions. There are two types of questions. The first, as shown in Figure 1-5, is the multiple-choice question. There is no limit to the number of answer options

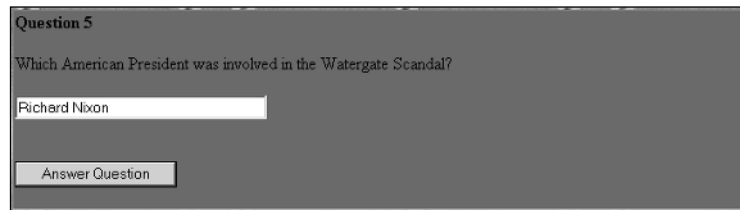
that we can specify for these types of questions: JavaScript handles it without the need for each question to be programmed differently.



A screenshot of a web interface for a quiz. It features a dark gray background. At the top left, the text "Question 1" is displayed. Below it, the question "What is Homer Simpson's favorite food?" is shown. There are four radio button options: "Fresh Salad", "Doughnuts", "Bread and Water", and "Apples". The "Fresh Salad" option is selected. At the bottom of the question area, there is a light gray button labeled "Answer Question".

Figure 1-5

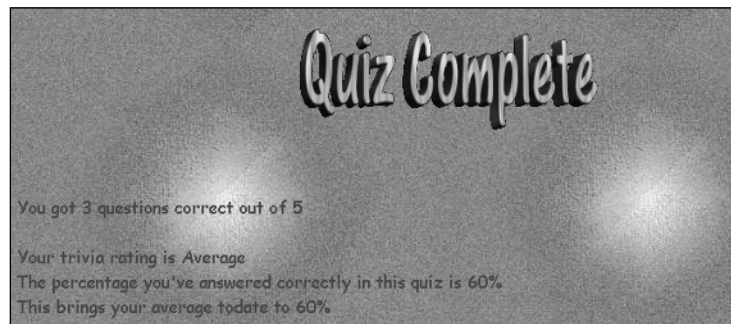
The second question style is a text-based one. The user types the answer into the box provided, and then JavaScript does its best to intelligently interpret what the user has written. For example, for the question shown in Figure 1-6, I have entered Richard Nixon, the correct answer. However, the JavaScript has been programmed to also accept R. Nixon, Nixon, Richard M. Nixon, and so on as a correct answer. We'll find out how in Chapter 8.



A screenshot of a web interface for a quiz. It features a dark gray background. At the top left, the text "Question 5" is displayed. Below it, the question "Which American President was involved in the Watergate Scandal?" is shown. There is a text input field containing the text "Richard Nixon". At the bottom of the question area, there is a light gray button labeled "Answer Question".

Figure 1-6

Finally, once the questions have all been answered, the final page of the quiz displays the user's results, as shown in Figure 1-7. This page also contains two buttons: one to restart the quiz and another to reset the quiz statistics for the current user.



A screenshot of a web interface showing the results of a quiz. The background is dark gray. At the top, the text "Quiz Complete" is displayed in a large, stylized, 3D font. Below this, the text "You got 3 questions correct out of 5" is shown. Further down, the text "Your trivia rating is Average" is displayed. Below that, the text "The percentage you've answered correctly in this quiz is 60%" is shown. At the bottom, the text "This brings your average to date to 60%" is displayed.

Figure 1-7

Ideas Behind the Coding of the Trivia Quiz

We've taken a brief look at the final version of the trivia quiz in action and will be looking at the actual code in later chapters, but it's worthwhile to consider the guiding principles behind its design and programming.

One of the most important ideas is code reuse. We save time and effort by making use of the same code again and again. Quite often in a web application, you'll find that you need to do the same thing over and over again. For example, we'll need to make repeated use of the code that checks whether a question has been answered correctly. You could make as many copies of the code as you need, and add this code to your page wherever you need it. However, this makes maintaining the code difficult, because if you need to correct an error or to add a new feature, you will need to make the change to the code in lots of different places. Once the code for a web application grows from a few lines in one page to many lines over a number of pages, it's quite difficult to actually keep track of the places where you have copied the code. So, with reuse in mind, the trivia quiz keeps all the important code that will need to be used a number of times in one place.

The same ideas go for any data you use. For example, in the trivia quiz we keep track of the number of questions that have been answered, and update this information in as few places as possible.

Sometimes you have no choice but to put important code in more than one place, for example, when you need information that can only be obtained in a particular circumstance. However, if you can keep it in one place, you'll find doing so makes coding more efficient.

In the trivia quiz, I've also tried to split the code into specific *functions*. We will be looking at JavaScript functions in detail in Chapter 3. In our trivia quiz, the function that provides us with a randomly selected question for our web page to display is in one place, regardless of whether this is a multiple-choice question or a purely text-based question. By doing this, we're not only just writing code once, but we're also making life easier for ourselves by keeping code that provides the same service or function in one place. As you'll see later in the book, the code for creating these different question types is very different, but at least putting it in the same logical place makes it easy to find.

When creating your own web-based applications, you might find it useful to break the larger concept, here a trivia quiz, into smaller ideas. Breaking it down makes writing the code a lot easier. Rather than sitting down with a blank screen and thinking, "Right, now I must write a trivia quiz," you can think, "Right, now I must write some code to create a question." I find this technique makes coding a lot less scary and easier to get started on. This method of splitting the requirements of a piece of code into smaller and more manageable parts is often referred to as "divide and conquer."

Let's use the trivia quiz as an example. Our trivia quiz application needs to do the following things:

- ☐ Ask a question.
- ☐ Retrieve and check the answer provided by the user to see if it's correct.
- ☐ Keep track of how many questions have been asked.
- ☐ Keep track of how many questions the user has answered correctly.
- ☐ Keep track of the time remaining if it's a timed quiz, and stop the quiz when the time is up.
- ☐ Show a final summary of the number of correct answers given out of the number answered.

These are the core ingredients for the trivia quiz. You may want to do other things, such as keep track of the number of user visits, but these are really external to the functionality of the trivia quiz.

Once you've broken the whole concept into various logical areas, it's sometimes worth using the "divide and conquer" technique again to break down the sub-areas into smaller chunks, particularly if the sub-area is quite complex or involved. As an example, let's take the "Ask a question" item from the preceding list.

Asking a question will involve the following:

- ❑ Retrieving the question data from where it's stored, for example from a database.
- ❑ Processing the data and converting it to a form that can be presented to the user. Here we need to create HTML to be displayed in a web page. How we process the data depends on the question style: multiple choice or text-based.
- ❑ Displaying the question for the user to answer.

As we build up the trivia quiz over the course of the book, we'll look at its design and some of the tricks and tactics that are used in more depth. We'll also break down each function as we come to it, to make it clear what needs to be done.

What Functionality Do We Add and Where?

How do we build up the functionality needed in the trivia quiz? The following list should give you an idea of what we add and in which chapter.

In Chapter 2, we start the quiz off by defining the multiple-choice questions that will be asked. We do this using something called an array, which is also introduced in that chapter.

In Chapter 3, where we talk about functions in more detail, we add a function to the code that will check to see whether the user has entered the correct answer.

After a couple of chapters of theory, in Chapter 6 we get the quiz into its first "usable" state. We display the questions to the user, and allow the user to answer these questions.

In Chapter 7, we enhance the quiz by turning it into what is called a "multi-frame application." We add a button that the user can click to start the quiz, and specify that the quiz must finish after all the questions have been asked, rather than the questions being repeated indefinitely.

In Chapter 8 we add the text-based questions to the quiz. These must be treated slightly differently from multiple-choice questions, both in how they are displayed to the user and in how the user's answers are checked. As we saw earlier, the quiz will accept a number of different correct answers for these questions.

In Chapter 9, we allow the user to choose the number of questions that he or she wishes to answer, and also whether he or she wants to have a time limit for the quiz. If users choose to impose a time limit upon themselves, we count down the time in the status bar of the window and inform them when their time is up.

Chapter 1

In Chapter 11, we will store information about the user's previous results, using cookies, which are introduced in that chapter. This enables us to give the user a running average score at the end of the quiz.

In Chapter 16, the quiz goes server-side! We move a lot of the processing of the quiz so that it occurs on the server before the page is sent to the user. We use the server to store information about the number of questions and time limit that the user has chosen, so that these values can be displayed to the user as the default values the next time he or she starts the quiz.

In Chapter 17, we change the quiz so that it gets its questions and answers from a database, making it easier to add and modify questions.

Summary

In this brief introduction to JavaScript you should have a feel for what JavaScript is and what it can do. In particular this chapter covered the following:

- ❑ We looked into the process the browser follows when interpreting our web page. It goes through the page element by element (parsing), and acts upon our HTML tags and JavaScript code as it comes to it.
- ❑ When developing for the web using JavaScript, there are two places where we can choose our code to be executed: server-side or client-side. Client-side is essentially the side on which the browser is running—the user's machine. Server-side refers to any processing or storage done on the web server itself.
- ❑ Unlike many programming languages, JavaScript requires just a text editor to start creating code. Something like Windows NotePad is fine for getting started, though more extensive tools will prove valuable once you get more experienced.
- ❑ JavaScript code is embedded into the web page itself along with the HTML. Its existence is marked out by the use of `<script>` tags. As with HTML, script executes from the top of the page and works down to the bottom, interpreting and executing the code statement by statement as it's reached.
- ❑ We introduced the online trivia quiz, which is the case study that we'll be building over the course of the book. We took a look at some of the design ideas behind the trivia quiz's coding, and explained how the functionality of the quiz is built up over the course of the book.