

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 web site.

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 sort 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 downwards. 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.

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 miss out code altogether, 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 into 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. 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 they 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++.

Perhaps this is a good point to dispel a widespread myth: JavaScript is not the same as 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, part of the Netscape Communicator suite of web applications. Initially, it was called LiveScript. However, since 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. In this book we'll make sure that our JavaScript code will work with the versions that come with both Netscape and Microsoft version 4 and later browsers. 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 the 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 like a text file. However, this is not how web pages are loaded when we browse web sites on the Internet. The Internet is really just one great big network connecting computers together. Web sites are a special service provided by particular computers on the Internet; the computers providing this service are said to **host web services** and are known as **web servers**.

Basically the job of a web server is to hold lots of web pages on its hard drive. Then, 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/shop keeper 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.

Towards 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 then see that web servers are not just dumb machines that pass pages back to clients, but in fact can do a bit of processing using JavaScript themselves. 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 web site 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, 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 to automate computer administration tasks. 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 use of JavaScript is interacting with your users, getting information from them and validating their actions. For example, say we wanted 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 have 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 their 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) and wondered how they manage that, then now's your chance to find out as this is another JavaScript trick. We'll demonstrate this later in the book. We'll also see how to create expanding menus that display a list of choices when a user rolls their mouse over them, another commonly seen JavaScript driven trick.

Tricks are OK up to a point, but even more useful can be small applications that provide a real service. Examples of the sort of things I mean are a mortgage seller's web site that has a JavaScript-driven mortgage calculator, or a web site about financial planning that includes a calculator that works out your tax bill for you. With a little inventiveness you'll find it's amazing 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, as that's where we need to add our JavaScript. There are also a number of very good tools specifically aimed at developing web-based applications, such as Microsoft's Visual InterDev. 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've got 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 your web site to be using. 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, and 6, and Internet Explorer versions 4.0 and 5.0. Wherever a piece of code does not work on any of these browsers, a note of this has been made in the text.

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, choose the **Advanced** tab. Check that the checkbox beside **Enable JavaScript** is checked. If not, then check it.

In Internet Explorer it is harder to turn off scripting. 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 (up to Chapter 15) you simply need to open the file from where it is stored on your hard drive. There are a number of ways to do this. One way you can do this in Internet Explorer is by choosing **Open** from the **File** menu, and clicking the **Browse** button to browse to where you stored the code. Similarly, in Netscape Navigator, you should choose **Open Page** from the **File** menu, and click the **Choose File** button, or in Netscape Navigator 6 choose **Open File** from the **File** menu.

The <SCRIPT> Tag and Your First Simple JavaScript Program

We've now talked around the subject of JavaScript for long enough; let's look at how we put it into our web page. 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 out 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, 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, we can't put them outside these areas, for example before the <HTML> tag or after the </HTML> tag, since anything outside these areas is not considered by the browser to be part of the web page and is ignored.

The <SCRIPT> tag has a number of attributes, but the most important one for us is the LANGUAGE attribute. As we saw above, 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">
```

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. By this I mean 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.

There are situations where JavaScript is not the default language, such as when script is run server-side (see Chapter 15), and in these situations 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.

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">
  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">
```

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 use of the `document` at the moment, as we'll be looking at it in depth later in the book.

Note that this line of code is an example of a JavaScript **statement**. Every line of code between the `<SCRIPT>` and `</SCRIPT>` tags is called a statement.

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 line of code. In practice, JavaScript is very relaxed about the need for semicolons and will usually be able to work out, when you start a new line, 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.

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've not looked at the order in which it works. When the browser loads in the web page it goes through it, rendering it line by line. 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">
  // Script block 1
  alert("First Script Block");
</SCRIPT>

<P>Paragraph 2</P>

<SCRIPT LANGUAGE="JavaScript">
  // Script block 2
  document.bgColor = "RED";
  alert("Second Script Block");
</SCRIPT>

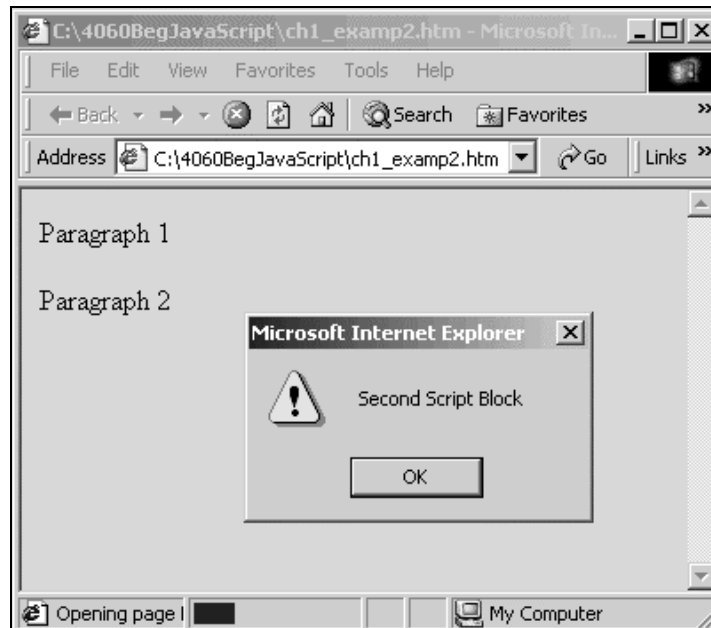
<P>Paragraph 3</P>

</BODY>
</HTML>
```

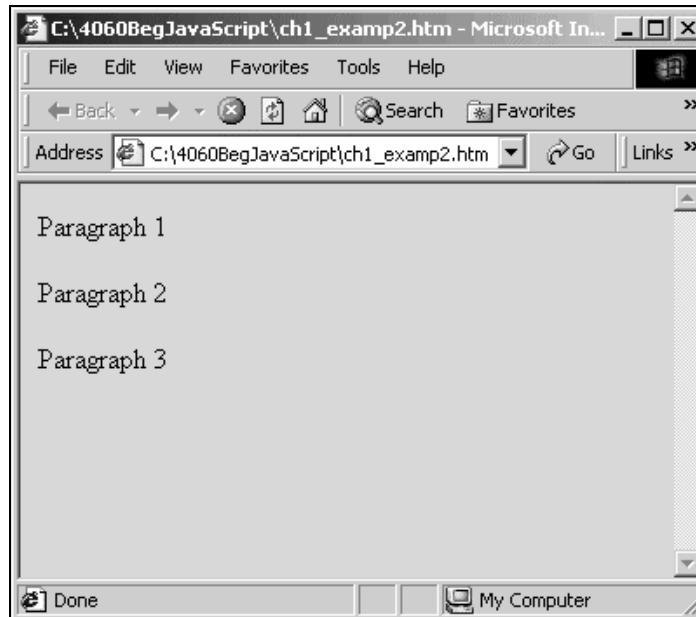
Save the file to your hard drive as `ch1_examp2.htm`, and then load it into your browser. When you load the page you should see the first paragraph, **Paragraph 1**, appear followed by a message box displayed by the first script block. The browser halts its parsing until you click the OK button. As you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



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 also displayed by the second script block.



Click OK and again the parsing continues, with the third paragraph, Paragraph 3, being displayed. The web page is complete.



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">
  // 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, since we can add explanations to our code that make it easier for us to work out 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 properly 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: it takes some information, processes it, and gives you a result. Functions makes life easier for us as programmers since 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, as 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">
  // 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. The second line of the script code we saw 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, then 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 example above that 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, unfortunately 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 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 upper case, 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 is really down to the browsers you think the majority of your web site'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+).

If we want our website to be professional, we need to somehow deal with older browsers. We could make sure our code is backwardly 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. What is meant by degrade gracefully is that, 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 their 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 re-direct 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 your pages work acceptably on as many browsers as possible.

Below is a table listing the different versions of JavaScript (and JScript) that Microsoft and Netscape browsers support. However, it is a necessary over simplification 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.

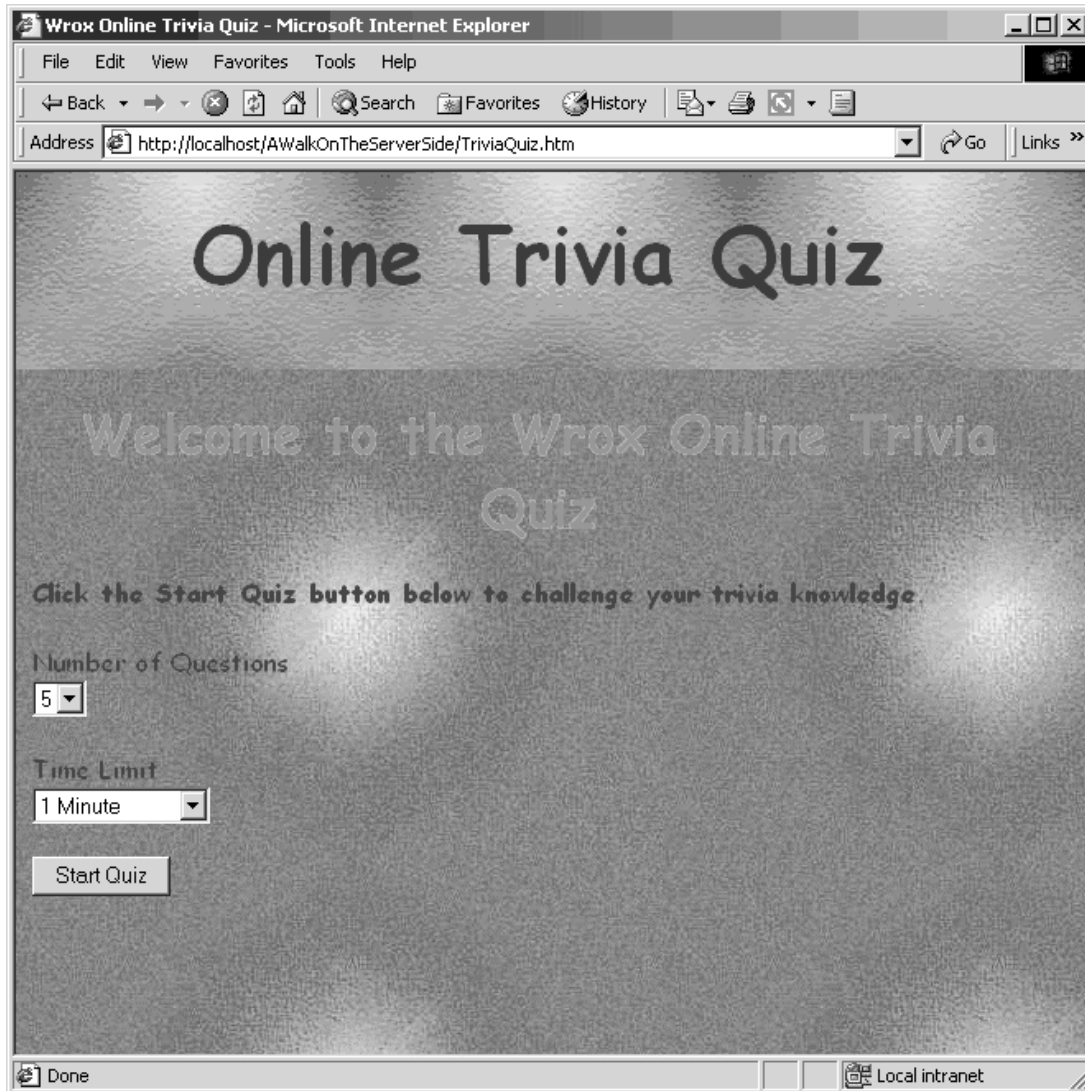
On a more positive note, the core of the JavaScript language does not vary too much between JavaScript versions; the differences are mostly useful extra features, which are nice-to-haves but often not essential. We'll concentrate on the core parts of the JavaScript language in the next few chapters.

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	-
JavaScript 1.2 (equivalent to JScript 3.0)	4.0 - 4.05	4.x
JavaScript 1.3	4.06+	-
JavaScript 1.4 (equivalent to JScript 5.0)	-	5.x
JavaScript 1.5	6.0	-

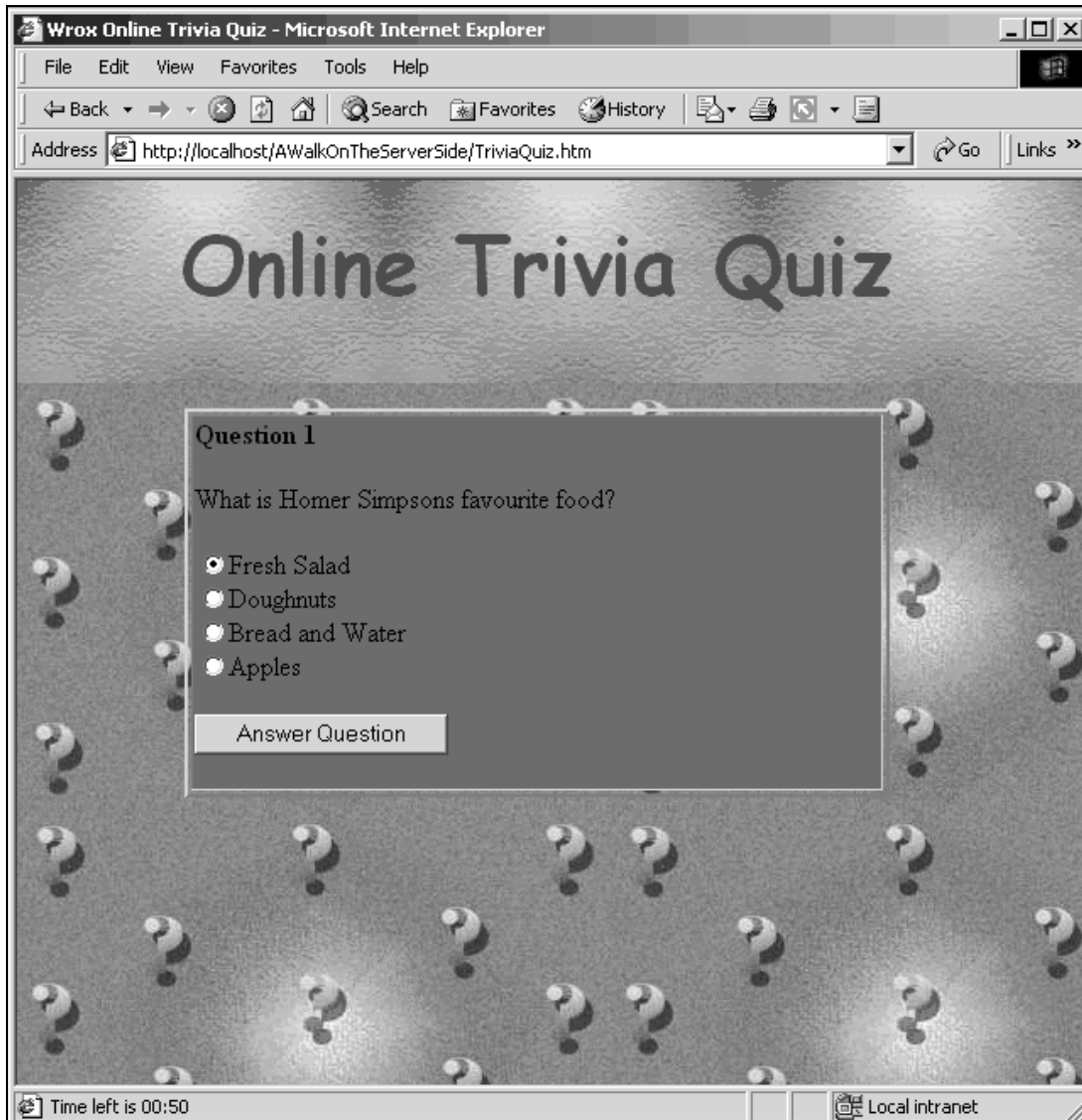
Introducing the Trivia Quiz

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.

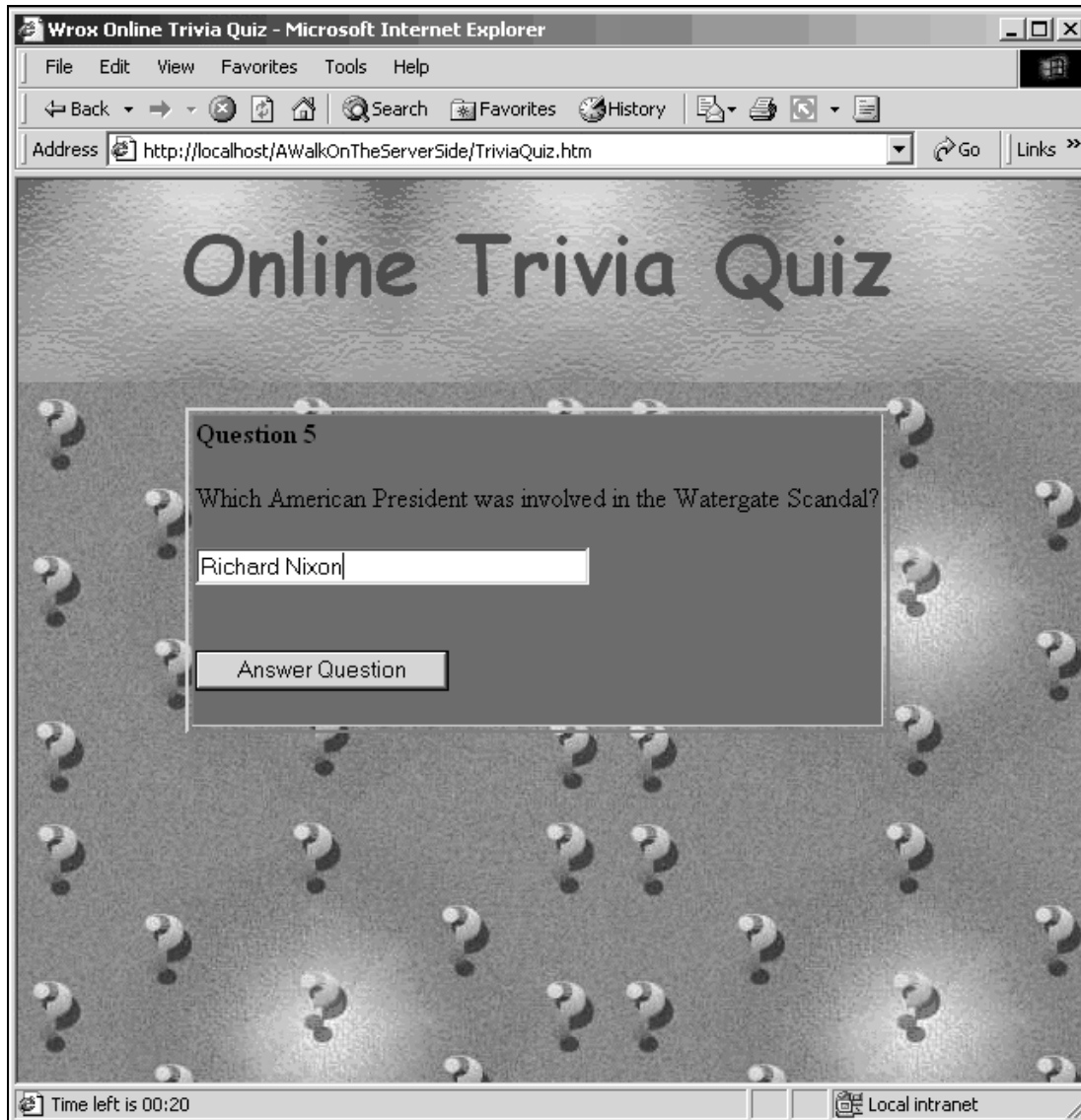
Let's take a look at what the quiz will finally look like. The main starting screen is shown below. Here the user can choose the number of questions that they want to answer and whether to set themselves a time limit. Using a JavaScript-based timer, we keep track of when the time that they have allotted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



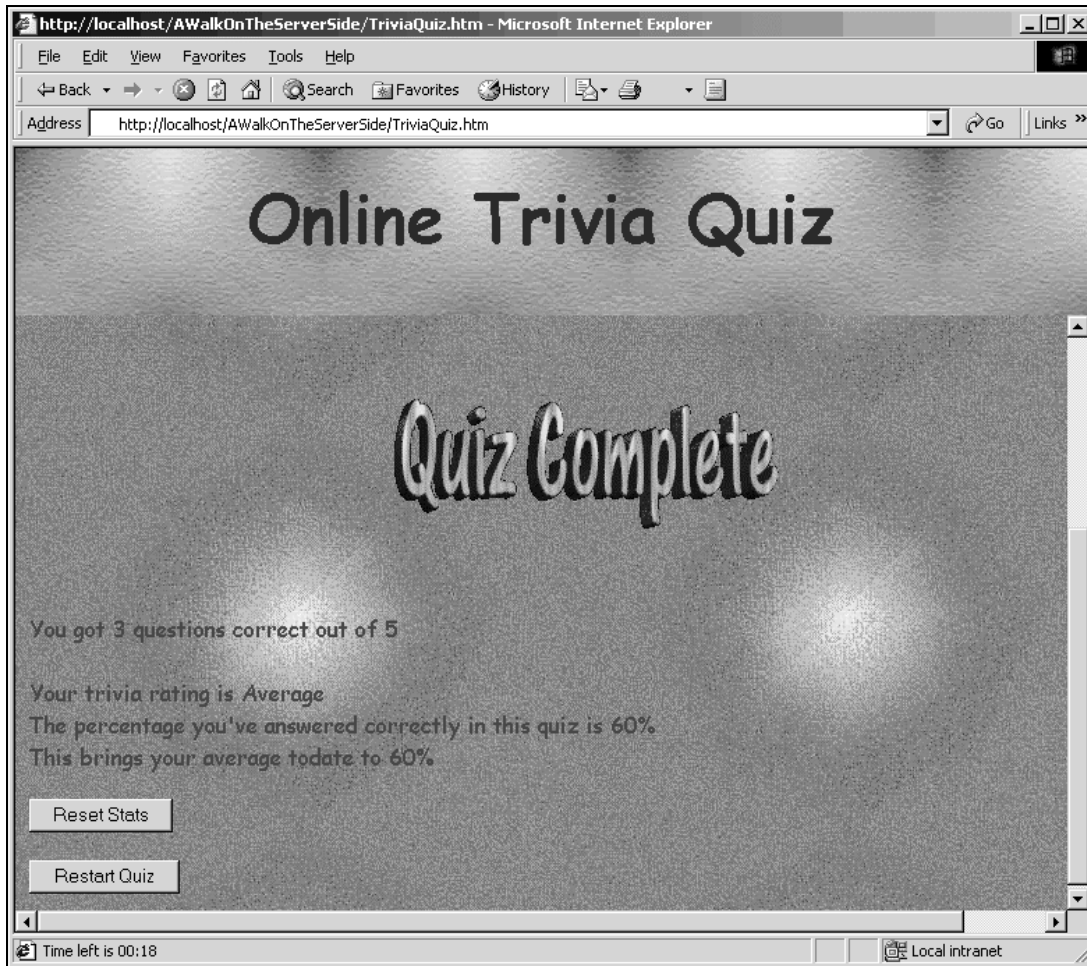
Having clicked the Start Quiz button, the user is faced with a random choice of question pulled from the database that we'll create to hold the trivia questions. There are two types of question. The first, as shown below, is the multiple-choice based question. There is no limit to the number of answer options that we can specify for these types of questions: the JavaScript handles it without the need for each question to be programmed differently.



The second question style is a text-based one. The user types the answer into a box provided and then JavaScript does its best to intelligently interpret what they have written. For example, for the question shown overleaf 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 on text manipulation.



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



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 considering what the guiding principles behind its design and programming are.

One of the most important ideas that we use is of 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 use of the code that checks if a question has been answered correctly many times. You could make as many copies of the code as you needed, 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, then 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 actually keeping 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 in one place, 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, then 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 down 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 got right
- 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. There may be other things that you want to do like keeping 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 above list.

Asking a question will involve:

- ❑ Retrieving the question data from where it is 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: multi-choice or text
- ❑ 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 if the user has entered the correct answer or not.

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 press 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 their answer is checked. As we saw above, 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 they wish to answer, and also whether they want to have a time limit for the quiz. If they 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.

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 15, 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 they start the quiz.

Finally, in Chapter 16, we change the way that the quiz gets the questions, from an array to a server-side database. This way, we can add new questions to the quiz more easily.

Summary

In this brief introduction to JavaScript you should have got 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 line by line (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 line by line 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.

