Introduction to JavaScript

Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke,
"Technology and the Future"

I n this chapter, we'll discover what makes JavaScript an object-based scripting language and take a look at the language's origins and history. Then we'll sink our teeth into the meat of JavaScript: objects, properties, methods, and events. We'll also

- see where JavaScript objects come from, when they are created, and how to access them.
- take a close look at the Document Object Model (DOM). What is it? And what does it have to do with JavaScript?
- examine the event model and how it makes JavaScript interactive.

What Is JavaScript?

In the introduction we discussed how JavaScript can be useful. But what *is* JavaScript exactly? JavaScript is an object-based scripting language modeled after C++. Java, too, was modeled after C++, which is one reason they look similar and are often confused as the same language. So while you're learning JavaScript, you're also learning a little of the basic command syntax for Java and C++. Keep in mind, however, that while Java is

a full-fledged, object-oriented programming language, developed by Sun Microsystems, complete with all the bells and whistles and complications inherent in object-oriented programming (OOP) languages, JavaScript is simply an **object-based** language. Most of Client-Side JavaScript's objects come from things found in Web pages such as images, forms, and form elements. That's one of the reasons JavaScript is considered an object-based language. More about that later.

side note

JavaScript is often said to be a sub-language of Java or derived from Java. That is simply not true. JavaScript is a programming language in its own right, developed by Netscape Communications. It is *not* a sub-language of Java.

OK, so JavaScript is a **scripting language**. "But what's a scripting language?" you ask. Scripting languages are programming languages that are generally easy to learn, easy to use, excellent for small routines and applications, and developed to serve a particular purpose. For instance, Perl (Practical Extraction and Report Language) began its life as a scripting language written for the express purpose of extracting and manipulating data and writing reports. JavaScript was written for the express purpose of adding interactivity to Web pages.

Scripting languages are usually **interpreted** rather than **compiled**. That means that a software routine, an interpreter, must translate a program's statements into **machine code**, code understandable by a particular type of computer, before executing them *every time the program is run*. Compiled languages, on the other hand, are translated into machine code and stored for later execution. When the compiled program is run, it executes immediately without further need of interpretation; it was interpreted into machine code when it was compiled. Because programs written in interpreted languages must be translated into machine code every time they are run, they are typically slower than compiled programs. However, this does not usually present a problem for the small applications for which scripting languages are generally used.

Being interpreted does have its advantages. One is platform independence. Because an interpreter performs the translation, you can write your program once and run it on a variety of platforms. All you need is the correct interpreter. In the case of JavaScript, the interpreter is built into Web browsers. Browsers are available for a variety of platforms and operating systems. Another advantage is that scripting languages are often loosely typed and more forgiving than compiled languages.

As a scripting language, JavaScript is easy to learn and easy to use. You can embed commands directly into your HTML code and the browser will interpret and run them at the appropriate time. JavaScript is also much more forgiving than compiled languages such as Java and C++. Its **syntax**, the special combination of words and symbols used by the language for programming commands, is simple and easy to read.

JavaScript is **loosely typed**. You don't have to declare the type of data that will be stored in a variable before you use it. If you decide to first store a number in a variable, then later

store a string in it, JavaScript will neither grumble nor complain, even if doing so *isn't* the best programming practice.

History

JavaScript was created by Brendan Eich of Netscape Communications and was first made available in 1995 as part of Netscape Navigator 2.0, the first JavaScript-enabled Web browser. Originally called LiveScript, JavaScript owes the Java part of its name to the popularity of Java, the cross-platform, object-oriented programming language created by Sun Microsystems. JavaScript was designed for the specific purpose (remember I said scripting languages are developed for a specific purpose) of extending the capabilities of Web browsers and providing Web developers with an easy means of adding interactivity to their Web sites.

Core JavaScript Version	ECMA Script Support	Client-Side JavaScript Version	Equivalent JavaScript Version	Browser Support		
				NN	IE	OP
1.0		1.0	1.0	2.0x	3.0x	—
1.1	version 1	1.1	2.0	3.0x,	3.02x[*]	3-0x–3.5x[*]
1.2		1.2[*]	3.0[**]	4.0x-4.05	4.0x[**]	—
1.3	version 2	1.3	5.0-5.1	4.06-4.7x	5.0x–5.1x[*]	4.0x–5.0x
1.4	version 3		5.5	5.0 (no release)	5.5x[*]	6.0x
1.5	version 3		5.6	6.0x-7.0x+	6.0x[*]	6.0x

ECMA = European Computer Manufacturers Association, NN = Netscape Navigator, IE = Internet Explorer, OP = Opera
[*] Not fully ECMA-262 compliant (current version listed in chart at that level)
[**] ECMA-262 version 1 compliant

Table 1.1   *Version History and Browser Support of JavaScript*

There are actually three flavors of JavaScript: Core JavaScript, Client-Side JavaScript, and Server-Side JavaScript. **Core JavaScript** is the basic JavaScript language. It includes the operators, control structures, built-in functions, and objects that make JavaScript a programming language (see Table 1.2). **Client-Side JavaScript** (CSJS) extends the JavaScript core to provide access to browser and Web document objects via the Document Object Model (DOM) supported by a particular browser. Client-Side JavaScript is, by far, the most popular form of JavaScript.

Objects	Operators	Functions	Control Statements
Array	+ - * / % ++ -- -n	Boolean()	? . . . :
Boolean	= += -= *= /= %=	escape()	break
Date	== != === !== < <= > >=	eval()	continue
Function	&& || !	isFinite()	do . . . while
global	& | ∧ ~ << >> >>>	isNaN()	for
Math	delete	Number()[NN4]	for . . . in
Number	new	parseFloat()	if
Object	this	parseInt()	if . . . else
RegExp	typeof	String()[NN4]	*label*:
String	void	unescape()	switch
			while
			with

Table 1.2   Core JavaScript

Another extension to Core JavaScript is **Server-Side JavaScript** (SSJS). A Netscape server-side technology that provides access to databases, it is a bit more complicated than its client-side sibling. SSJS, like CSJS, is embedded directly within HTML documents. However, in order to increase execution speed, an important concern when assembling Web pages on the server before delivering them to the Web browser, HTML documents that contain SSJS must be precompiled into JavaScript byte code. When a Web page that contains Server-Side JavaScript is requested, the server retrieves the document, executes the appropriate byte code on the server, accesses databases and other resources as necessary, and serves up the results as plain vanilla HTML to the browser. Because it returns plain HTML, SSJS can serve any type of Web browser and runs on any SSJS-enabled Web server. Client-Side JavaScript requires a JavaScript-enabled Web browser. Most of today's browsers are JavaScript-enabled.

In June 1997, the European Computer Manufacturers Association (ECMA) announced a new cross-platform scripting language standard for the Internet known as ECMA-262 or **ECMAScript**. The standard is based on Netscape's Core JavaScript 1.1. ECMAScript received International Organization for Standardization (ISO) approval as international standard ISO/IEC 16262 in April 1998. After some small editorial changes, the ECMA General Assembly of June 1998 approved a second edition of ECMA-262 in order to keep it fully aligned with ISO/IEC 16262. A third edition was approved in December 1999; it includes many improvements.

The subject of this book is Client-Side JavaScript, which includes Core JavaScript. However, in this text we'll refer to it simply as "JavaScript."

Objects, Properties, and Methods

Because JavaScript is an object-based language, we will be working with, you guessed it, **objects**, as well as their properties and methods. So what *is* an object? Well, what's an object in the real world? It's an item, a thing. And as an item or thing, it has attributes or **properties** that describe it and make it unique. An object also has **methods**: actions you can perform with the object or on the object.

For instance, take your writing pen. It's an object. It has a case color on its outside, a particular ink color, and a type such as ballpoint, fountain pen, rollerball, etc. These pen attributes or *properties* describe your particular pen.

Now what is a pen good for? For writing, of course. Writing is an action you can perform *with* your pen. So we could say that your pen has a write *method*. What other actions can be performed with or on your pen? When the ink runs out, you can refill your pen. So we could say that your pen has a refill method. In this case, refilling is an action performed *on* your pen. Methods are, therefore, actions that we can perform with or on an object.

Another way to think of objects and their properties and methods is in terms of the parts of speech. In this case, objects are nouns, properties are adjectives because they describe the noun, and methods are verbs because they specify what the noun does.

JavaScript uses **dot notation** to refer to an object and its associated properties and methods. For instance, in dot notation, we refer to your pen as simply

```
pen
```

To reference your pen's ink color property we use the name of the object followed by a dot (period) and the name of the property. Notice that there are no spaces in the object and property names.

```
pen.inkColor
```

Changing the value of an object's property is often just the simple matter of assigning a new value to it. For instance, to change your pen's ink color, we could use a statement like this:

```
pen.inkColor = "blue"
```

The dot notation for methods is slightly different. While we still reference the name of the object first, followed by a dot and the method name, we end with a set of parentheses. The parentheses are meant to hold any arguments or parameters that provide data to or modify the performance of a method. For example, we need to be able to tell our pen what to write, so we send in an argument representing the text we want to write:

```
pen.write("Hello")
```

Now let's consider a real-world JavaScript example. Remember I said in the introduction that JavaScript derives most of its objects from Web documents? In fact, one of the most important objects in JavaScript is the `document` object itself.

Many HTML tag attributes can be accessed as object properties with JavaScript. Most, but not all, object properties have the same name as their HTML tag attribute equivalent. For instance, the appropriate attribute of the <body> tag for setting the background color is `bgcolor`. The equivalent `document` property name in JavaScript is `bgColor` (same name, different capitalization). Keep in mind that while HTML is **case insensitive**, JavaScript is **case sensitive**. While to HTML `BGCOLOR`, `bgColor`, and `bgcolor` are all the same, to JavaScript they are not the same at all!

Remember I said, "Most, but not all, object properties have the same name as their HTML tag attribute equivalent." The attribute of the <body> tag that specifies the document's text color is `text`. The equivalent document property is `fgColor`, which is short for "foreground color." HTML and JavaScript are referring to the same thing, only HTML uses the term "text," while JavaScript uses the term "fgColor." This happens from time to time. You'll find Appendix A, "JavaScript Object and Language Reference," to be quite useful in figuring out property names.

Let's get back to that real-world example I promised you. To refer to the `document` object itself in JavaScript dot notation, we simply write

```
document
```

To refer to its background color we write

```
document.bgColor
```

We can also write directly to the document using the document's `write` method:

```
document.write("Hello")
```

Try it for yourself. Open your favorite HTML editor or text editor and type the following code:

side note

Technically, `document` is a property of the `window` object. Its formal dot notation is `window.document`. However, JavaScript assumes when you use the shorthand, `document`, that you must be referring to the current window, that is, the window in which the document resides. Using the shorthand saves us a lot of typing, but there are times when it is critical to specify the window formally. We'll discuss those instances in Chapter 11, "Windows and Frames." For now, we'll continue to use the shorthand, `document`, instead of the formal, `window.document`, to reference the `document` object.

```
1    <html>
2    <head>
3    <title>Script 1.1: Using the Write Method</title>
4    </head>
5    <body bgcolor="white" text="black">
6    <script language="JavaScript" type="text/javascript">
7         document.write("Hello")
8    </script>
```

```
 9    </body>
10    </html>
```

Script 1.1   Using the `write` Method

Save your document as a text file with an .html extension and open it in your browser.
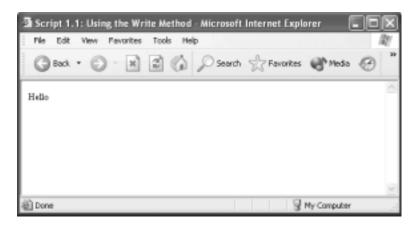The results should look similar to this:

Figure 1.1   Results of Script 1.1

You can even use JavaScript to write HTML. What may seem confusing at first is that
you write HTML tags like other strings in JavaScript, delimited by quotation marks.
JavaScript doesn't understand HTML codes per se. The way it works is that you can use
JavaScript to write the HTML code for you, *then* the HTML interpreter in the browser
parses the HTML and applies your markup. You can also use JavaScript to write a <style>
tag and stylesheet rules in the head of a document. Cool, huh?

To see this in action, let's modify Script 1.1 to write the greeting as a centered heading
with the <h1> tag.

```
 1    <html>
 2    <head>
 3    <title>Script 1.2: Using the Write Method to Write HTML</title>
 4    </head>
 5    <body bgcolor="white" text="black">
 6    <script language="JavaScript" type="text/javascript">
 7    document.write("<h1 align=center>Hello</h1>")
 8    </script>
 9    </body>
10    </html>
```

Script 1.2   Using the write Method to Write HTML

Here's what it looks like:

Figure 1.2   Results of Script 1.2

Now let's change the document's background and foreground colors from white and black (as set on the <body> tag) to blue and white. Insert the following lines before line 8 of Script 1.2 and change the title:

```
document.bgColor = "blue"
document.fgColor = "white"
```

Your script should now look like this:

programmer's tip

It's OK to place spaces around the equals sign. It doesn't change the way the script works, but it does improve readability.

```
1    <html>
2    <head>
3    <title>Script 1.3: Changing Background &amp;
4         Foreground Colors</title>
5    </head>
6    <body bgcolor="white" text="black">
7    <script language="JavaScript" type="text/javascript">
8         document.write("<h1 align=center>Hello</h1>")
9         document.bgColor = "blue"
10        document.fgColor = "white"
11   </script>
12   </body>
13   </html>
```

Script 1.3   Changing the Background and Foreground Colors

Save the changed file with a new name and view it in your browser. This time, the document's background color will change to blue and its foreground color to white, like this:

Figure 1.3   Results of Script 1.3

Cool, huh?

Let's modify the script a little more. This time we'll examine the value of the document's `bgColor` and `fgColor` properties before and after our changes. Again, we can have JavaScript write the HTML to mark up the text. The document's `write` method can accept multiple string parameters; we simply have to separate them with commas, like this:

```
document.write(string1, string2, string3, ... stringN)
```

If you have really long strings to write, you can place them on separate lines; just always break at a comma, like this:

```
document.write(reallyReallyReallyReallyLongString1,
               reallyReallyReallyLongString2,
               string3, ... stringN)
```

This way, you can use a single `document.write` statement to write several lines of text.

Insert the following two lines of code before line 9 of Script 1.3:

```
document.write("<i>Original bgcolor: ", document.bgColor, "</i><br>")
document.write("<i>Original fgcolor: ", document.fgColor, "</i><br>")
```

Keep in mind, however, that you cannot simply let a string of characters wrap onto more than one line, like this:

```
document.write("<i>A Really Long Word: Supercalifragilistic
expialidocious</i>")
```

This would cause an error. You have two choices to fix it: keep the text all on one line or break the string into two strings, like this:

```
document.write("<i>A Really Long Word: Supercalifragilistic",
"expialidocious</i>")
```

This is perfectly legal and will output exactly as you intended.

Another important thing to keep in mind: Do *not* place quotation marks around object, property, or method references. Script 1.3 shows an example of how they should be written in lines 9 and 10: `document.bgColor` and `document.fgColor`. Both are property references and should not be quoted.

Insert the following two lines of code after line 10 of Script 1.3:

```
document.write("<b>New bgcolor: ", document.bgColor, "</b><br>")
document.write("<b>New fgcolor: ", document.fgColor, "</b><br>")
```

Your new script should look like this:

```
1    <html>
2    <head>
3    <title>Script 1.4: Changing Background &amp;
4         Foreground Colors</title>
5    </head>
6    <body bgcolor="white" text="black">
7    <script language="JavaScript" type="text/javascript">
8         document.write("<i>Original bgcolor: ",
9             document.bgColor, "</i><br>")
10        document.write("<i>Original fgcolor: ",
11            document.fgColor, "</i><br>")
12        document.write("<h1 align=center>Hello</h1>")
13        document.bgColor = "blue"
14        document.fgColor = "white"
15        document.write("<b>New bgcolor: ",
16            document.bgColor, "</b><br>")
17        document.write("<b>New fgcolor: ",
18            document.fgColor, "</b><br>")
19   </script>
20   </body>
21   </html>
```

Script 1.4   *Changing the Background and Foreground Colors*

Notice that each of the new `document.write` statements has three parameters; they're separated by commas. The document's `write` method allows you to send multiple parameters, each representing a piece of text to write to the document. For instance, in line 8 of Script 1.4, the browser is told to write the text `"<i>Original bgcolor: "`, followed by the current (original) value of the `bgColor` property, followed by a closing `</i>` and a `<br>` tag. Without the `<br>` tags, all the text would write on a single line. Remember, HTML ignores white space, so writing the JavaScript statements on different lines doesn't automatically get the content on different lines.

Now save your modified script with a new name and run it. The results should look similar to this:

Figure 1.4   Results of Script 1.4

Notice that the colors are listed in hexadecimal: #ffffff is the hexadecimal equivalent of white, #000000 is black, and #0000ff is blue. White was the value initially assigned to the `bgColor` property of the document by the HTML in line 6. The values listed after "Hello" are the values we assigned to the `bgColor` and `fgColor` properties with JavaScript.

Table 1.3 summarizes our discussion of objects, properties, and methods.

	Description	English Analogy	Real-world Example	JavaScript Example
Object	An item or thing.	noun	`pen`	`document`
Property	An attribute that describes an object.	adjective	`pen.inkColor`	`document.bgColor`
Method	An action that can be performed with or on an object.	verb	`pen.write()`	`document.write()`

Table 1.3   Summary of Objects, Properties, and Methods

Where Do JavaScript Objects Come From?

JavaScript objects originate in one of four places. Some are built into the language, like `Math`, `String`, `Date`, and `Array`; remember Core JavaScript? (See Table 1.2.) Most come from Web documents and are made available to Client-Side JavaScript via the Document Object Model (DOM). We've also seen that many object properties come from Web docu-

ments, often the attributes of HTML tags. A few objects come from the browser, such as the `navigator`, `location`, and `history` objects also made available to Client-Side JavaScript by the DOM. Lastly, we, as programmers, can create our own custom objects. Because most of JavaScript's objects come from Web documents, let's take a closer look at those first.

When a Web document contains an image, specified by an <img> tag in HTML, that document is said to "contain an `image` object." If it has a form, specified by a <form> tag, it contains a `form` object. It also may have paragraphs, headings, blockquotes, divisions, etc., but only the very latest browsers allow us to access or manipulate those objects, and they're still working on doing it the same way in compliance with Web standards. So what determines which objects we can manipulate? The Document Object Model implemented by a particular browser determines which objects we can access and manipulate in that browser.

The Document Object Model

Think of the **Document Object Model** as a template built into a browser that specifies all of the objects and properties in a Web document that the browser can identify and allow you to access and manipulate, plus all of the methods you can perform with or on those objects. The Document Object Model represents *possibilities*.

Unfortunately, there's currently only a smattering of agreement among the major browser vendors, Netscape, Microsoft, and Opera, as to what possibilities to support and how to access them. The situation *is* improving with each new browser release as the major browser vendors adopt more of the current World Wide Web Consortium (W3C) DOM standard as well as the W3C's HTML and Cascading Style Sheets (CSS) standards. Pressure from standards support groups like The Web Standards Project (WaSP) have helped encourage browser vendors to comply with W3C standards.
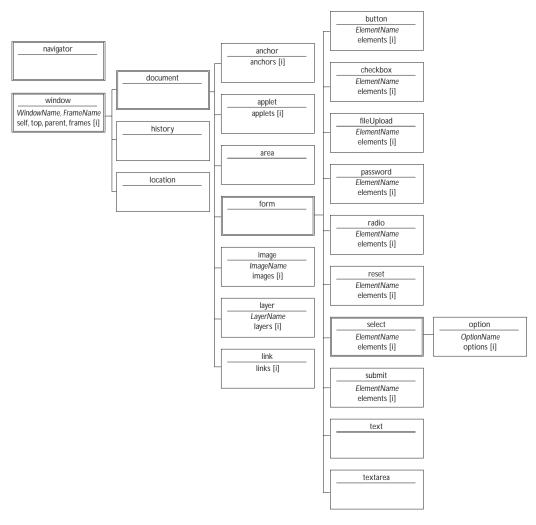
Figure 1.5   Cross-Browser DOM (JavaScript 1.1)

The above DOM illustration represents the DOM common to all three major browsers: Netscape Navigator, Microsoft Internet Explorer, and Opera. The newest browsers each support many more objects than those shown here, but they do so independently of or inconsistently with the other browser companies. See Appendix E, "Evolution of the Document Object Model," for a detailed description.

In the meantime, there are still plenty of objects for us to work with. While it can be frustrating sifting through all the differences and exceptions, there is a set of objects that we can safely rely on support for in all JavaScript-enabled browsers: those listed in Figure 1.5. Our initial focus in this book will be on those objects and their associated

properties and methods. As you become more familiar with JavaScript, I will introduce additional objects, properties, and methods and specify which browsers support them. Sound good? Then let's go see how objects are created.

When Are Objects Created?

The items contained in a Web document become objects in the browser's memory as the browser loads and interprets the HTML code that defines them. Notice that the DOM is arranged in a hierarchical way with window as the highest-level object. If you think about it a moment, you'll realize that every Web document is displayed or contained within a browser window. In essence, every object in a Web document is contained within some other object, with the exception of the window and navigator objects. The window object is the topmost or outermost container, and the navigator object, which represents the browser, is on the same level. Netscape characterizes this as an "**instance hierarchy**."

Netscape originally called its DOM an "instance hierarchy" because its objects come into being the *instant* that the HTML code or JavaScript code that defines them is read and interpreted by the browser. Each Web document, therefore, creates a different and unique instance hierarchy. While the Document Object Model supported by a particular browser provides *possibilities*, the instance hierarchy specifies the *actual objects* that are created in the browser's memory when a particular Web document loads. If you stop a Web document's loading halfway through, only those objects already read and interpreted by the browser will be represented in its instance hierarchy.

Let's take a look at a simple HTML document, listed in the first column, and the instance hierarchy the browser creates in memory as it loads the document. We'll use the DOM in Figure 1.5 as our guide.
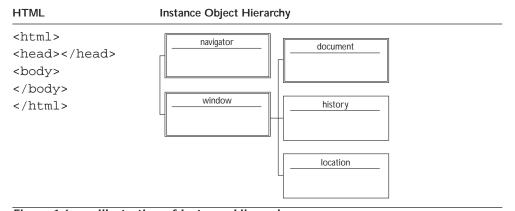

Figure 1.6   Illustration of Instance Hierarchy

OK, so this Web document doesn't really contain anything. If you preview it in your browser, you'll see only a blank page. Still, notice that this document does have an instance hierarchy. It's turned on its side to make it easy to see the associated dot notation:
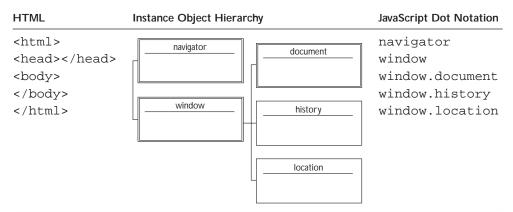
HTML	Instance Object Hierarchy	JavaScript Dot Notation

```
<html>
<head></head>
<body>
</body>
</html>
```



```
navigator
window
window.document
window.history
window.location
```

Figure 1.7   Illustration of Instance Hierarchy with Dot Notation

Dot notation dictates that you write the object name, followed by a dot and the object's property. Read the instance object hierarchy diagram above, left to right, inserting dots as needed and you get the dot notation listed in the last column. The `navigator` object represents the browser and is one of two top-level objects. Its name is rather Netscape biased, "browser" would be a non-biased name, but then Netscape was the company that created the JavaScript language, so I think that entitles them to some liberties. The `window` object is the other top-level object.

The `document` object is itself a property of the `window` object. This makes sense, because all documents in a Web browser are displayed in a window. The `history` object, also a property of the `window` object, represents the history list in the window. The history list contains the most recent pages you've visited during the current browser session, those that you can get to with the Back and Forward buttons. The `location` object, also a property of the window, represents the location of the current document (usually shown in the location bar, also known as the address bar) in the browser window.

Remember how we examined the contents of the document's `bgColor` and `fgColor` properties before and after we changed them in Script 1.4 using `document.write` statements? Let's use that same method to examine some of the properties of the objects listed in Figure 1.7 above.

Flip to Appendix A at the back of this book. You'll see a complete JavaScript object reference. You're going to be referring to this frequently, so you might want to stick a bookmark in it. It lists all of the objects, properties, and methods currently supported by the JavaScript language. Look up the `navigator` object. It will be listed in the "Client-Side JavaScript Objects" section.

navigator		
appCodeName	javaEnabled()[JS1.1]	No event handlers
appName	plugins.refresh()	
appVersion	preference(*prefName*[,*newValue*])[JS1.2]	
language[JS1.2]	savePreferences()[JS1.2]	
mimeTypes[ ][JS1.1]	toString()	
platform[JS1.2]		
plugins[ ][JS1.1]		
userAgent		

Figure 1.8   `navigator` *Object Listing from Appendix A*

In the client-side section, each object has three columns listed under it. The first column lists all of the properties supported by that object as provided by the JavaScript language itself. The second column lists the methods that apply to that object, and the third lists the event handlers that can be used to handle activity on the object.

We're going to use the document's `write` method (look it up, you'll find it listed in the second column under the document object) to display some of the `navigator` object's properties in Script 1.5, as well as those of the `window`, `history`, and `location` objects. Look up each object yourself and examine its properties. We'll use the dot notation listed in Figure 1.7 as a starting point and append each object with a dot and a property. For instance, to access the `userAgent` property of the `navigator` object, start with

```
navigator
```

and follow with a dot and the property name:

```
navigator.userAgent
```

Watch the case—remember JavaScript is case sensitive. Create a new document and type in the following code, or copy Script 1.5 from your CD-ROM. Typing the code yourself is good experience and highly recommended.

```
1    <html>
2    <head>
3    <title>Script 1.5: Examining Property Values
4            Using document.write</title>
5    </head>
6    <body>
7
8    <script language="JavaScript" type="text/javascript">
9      document.write("<h2>Some navigator Properties</h2>")
10     document.write("appName: ", navigator.appName, "<br>")
11     document.write("appVersion: ", navigator.appVersion, "<br>")
12     document.write("userAgent: ", navigator.userAgent, "<br>")
13
```

```
14    document.write("<h2>Some window Properties</h2>")
15    document.write("innerHeight: ", window.innerHeight, "<br>")
16    document.write("innerWidth: ", window.innerWidth, "<br>")
17    document.write("location: ", window.location, "<br>")
18
19    document.write("<h2>Some history Properties</h2>")
20    document.write("length: ", window.history.length, "<br>")
21
22    document.write("<h2>Some location Properties</h2>")
23    document.write("href: ", window.location.href, "<br>")
24    document.write("protocol: ", window.location.protocol, "<br>")
25  </script>
26  </body>
27  </html>
```

Script 1.5   Examining Property Values Using `document.write`

Save the file to your floppy or Zip disk and open it in a browser. Your results my differ from those shown below depending on what browser you use and where you saved the file. Lines 10 through 12 of Script 1.5 write three properties of the `navigator` object to the document: `appName`, `appVersion`, and `userAgent`. The `appName` property indicates the browser's application name, `appVersion` indicates the browser's version number, and `userAgent` represents the user-agent header sent by HTTP from the browser to the Web server.
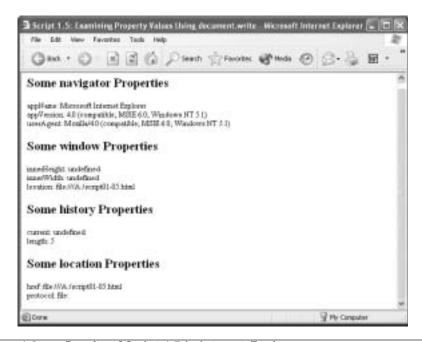

Figure 1.9   Results of Script 1.5 in Internet Explorer

The window properties `innerHeight` and `innerWidth` (lines 15 and 16) are supposed to specify the inner dimensions of the window. Unfortunately, Internet Explorer's DOM doesn't support them and so wrote "undefined." The window's `location` property (line 17) *is* supported by Internet Explorer (IE) and shows the URL to the file currently being displayed in the window. The `history` property, `length`, indicates the total number of items in the history list (line 20). Notice that IE does not support the current property. Finally, the `location` properties, `href` and `protocol`, represent the URL of the current document being viewed and its protocol (lines 23 and 24).

Let's see how this same document looks in a different browser. How about Netscape Navigator:

Figure 1.10   Results of Script 1.5 in Netscape Navigator

It's strange that this browser shows version 5 as its `appVersion` when we're using Navigator 7.0. It just goes to show that the `appName`, `userAgent`, and `appVersion` properties don't always report what you expect. Still, all three are often used to perform browser detection.

Look at the properties of the `location` object listed in Appendix A. See how you can access any part of a location's URL? Here's what each property that's a part of the URL means:

- ✍ hash—indicates the part of the URL that follows the hash mark or pound sign, that is, a named anchor reference.
- ✍ host—lists the server name, subdomain, and domain name.
- ✍ hostname—provides the full hostname of the server, including the server name, subdomain, domain, and port number.
- ✍ href—the entire URL.
- ✍ pathname—the path part of the URL, including the file name.
- ✍ port—the port number, if specified in the URL. If not, returns nothing.
- ✍ protocol—just the protocol portion of the URL, such as http:, ftp:, or file:.

You can check these out for yourself using Netscape Navigator and the javascript: pseudo-protocol. (This only works with Netscape Navigator, so if you don't have a copy, I highly recommend that you download one now.) Type a URL in the location bar. I'm going to visit Yahoo!, then click on Help and then Shopping so I get a nice long URL that includes a path as well as a domain name in the location bar. You don't have to visit Yahoo!, just visit the interior page of some Web site so you have a URL that has both a domain name and a path showing in the location bar.
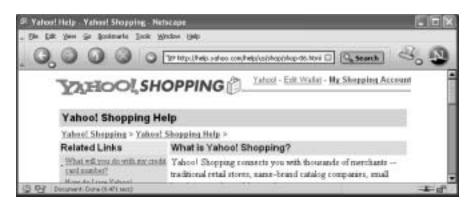

Figure 1.11   Visiting a Web Site's Interior Page

OK, now the fun part begins. Type

```
javascript: location.href
```

in the location bar of your Netscape Navigator browser. The javascript: pseudo-protocol allows you to run a JavaScript statement directly in your browser. You should get something like this after you press the Enter key.

*Figure 1.12   Using the javascript: Pseudo-Protocol
to Display the Location's `href` Property*

As you can see, the href property displays the entire URL. Let's try another. Click on the
Back button of your browser to return to the Web page you were viewing. Now type

    javascript: location.protocol

in the location bar and press the Enter key.

Figure 1.13   Viewing the `protocol` Property of the `location` Object

This time only the protocol portion of the location is displayed. Let's look at the
hostname. Click on the Back button, then type

    javascript: location.hostname

in the location bar and press Enter.

Figure 1.14   Viewing the `hostname` Property of the `location` Object

Now one more. Click on the Back button and type the following:

```
javascript: location.pathname
```

Press the Enter key.

Figure 1.15   Viewing the `pathname` Property of the `location` Object

Cool, huh?

You can change the current location, that is, you can open a new location or URL in the current window, by modifying the `location` object's `href` property, like this:

```
location.href = newURL
```

Let's try it. In the location bar, type the following:

```
javascript: location.href = "http://google.com"
```

Press Enter. Navigator should take you to Google.

Figure 1.16   Changing the Location

Cool, huh? You'll use this a lot in scripts to open new documents in a window.

programmer's tip

You can save JavaScript statements using the javascript: pseudo-protocol as bookmarks and call them at any time to help you examine the results of scripts and assist with debugging. Here's how:

1. Type "javascript:" followed by the statement you'd like to invoke in the location bar.
2. From the Bookmarks menu, choose Add to Bookmarks or File Bookmark. I recommend the latter; then you can file them all in the same folder. I put mine in a bookmark folder named "JavaScript Statements."
3. Now you can access them at any time from your Bookmark list, like this:

Nice little tool, huh?

OK, let's continue our discussion about the instance object hierarchy and how objects are created with another example. Let's modify the original HTML document shown in

Figure 1.7 by adding an image. You can find the image in the images folder on the CD-ROM that accompanies this book.
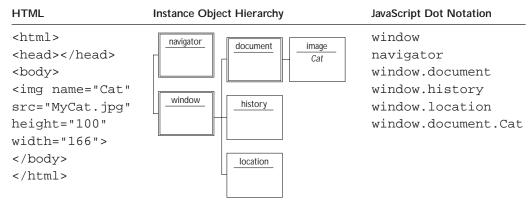
HTML	Instance Object Hierarchy	JavaScript Dot Notation

```
<html>
<head></head>
<body>
<img name="Cat"
src="MyCat.jpg"
height="100"
width="166">
</body>
</html>
```

```
window
navigator
window.document
window.history
window.location
window.document.Cat
```

Figure 1.17   Illustration of Instance Hierarchy with Image Added

Notice that the instance hierarchy now contains an image; that image object's name is Cat. To see what properties an image object has, turn to Appendix A, "JavaScript Object and Language Reference," and look up the image object in the "Client-Side JavaScript Objects" section. Notice how the properties of an image object roughly correspond to the attributes of an <img> tag?

Property	Value
window.document.Cat.name	Cat
window.document.Cat.src	MyCat.jpg
window.document.Cat.height	100
window.document.Cat.width	166

Table 1.4   Property Values for *image* Object Shown in Figure 1.17

Let's write those properties out.

```
1    <html>
2    <head>
3    <title>Script 1.6: Examining Properties of the Cat Image
4         Object</title></head>
5    <body>
6    <img name="Cat" src="images/MyCat.jpg" height="100" width="166">
7
8    <script language="JavaScript" type="text/javascript">
9      document.write("<h2>Some Properties of the Cat Image
10        Object</h2>")
11      document.write("Name: ", window.document.Cat.name, "<br>")
```

```
12    document.write("Src: ", window.document.Cat.src, "<br>")
13    document.write("Width: ", window.document.Cat.width, "<br>")
14    document.write("Height: ", window.document.Cat.height, "<br>")
15  </script>
16  </body>
17  </html>
```

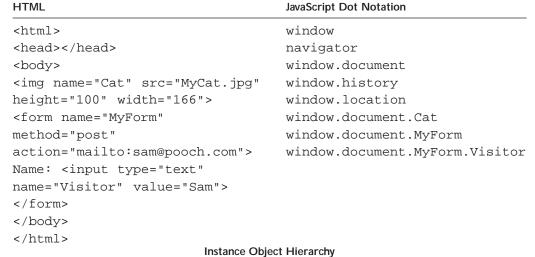**Script 1.6** *Examining Properties of the* `Cat image` *Object*

Lines 11 through 14 print the `name`, `src`, `width`, and `height` properties of the `Cat` `image` object. Notice that we accessed the `Cat image` object by its name, the name given it with the HTML name attribute on the <img> tag. A document can contain many images. The best way to distinguish between them is to give each a name. Then you can use that name to easily access that particular `image` object. I like to capitalize the names of objects I create with HTML so I can easily distinguish them from those I create with JavaScript. Then when I see a capitalized object name, I know immediately to look in the HTML document for that object.

Here's what the script looks like when run in a browser:

Figure 1.18 *Results of Script 1.6*

Now let's add a form and a text element to further illustrate this idea of accessing an object by its name as provided by the HTML element's name attribute.
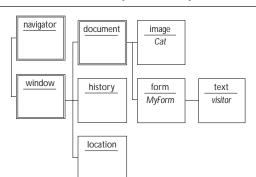
HTML	JavaScript Dot Notation
`<html>`	`window`
`<head></head>`	`navigator`
`<body>`	`window.document`
`<img name="Cat" src="MyCat.jpg"`	`window.history`
`height="100" width="166">`	`window.location`
`<form name="MyForm"`	`window.document.Cat`
`method="post"`	`window.document.MyForm`
`action="mailto:sam@pooch.com">`	`window.document.MyForm.Visitor`
`Name: <input type="text"`	
`name="Visitor" value="Sam">`	
`</form>`	
`</body>`	
`</html>`	

Instance Object Hierarchy

Figure 1.19   Illustration of Instance Hierarchy with Form and Text Element Added

The form has the following properties:

Property	Value
`window.document.MyForm.name`	MyForm
`window.document.MyForm.method`	post
`window.document.MyForm.action`	mailto:sam@pooch.com

Table 1.5   Property Values for Form Object Shown in Figure 1.19

The text box has these properties:

Property	Value
`window.document.MyForm.Visitor.name`	Visitor
`window.document.MyForm.Visitor.type`	text
`window.document.MyForm.Visitor.value`	"Sam" or whatever value is typed in the box

Table 1.6   *Property Values for Text Object Shown in Figure 1.19*

Let's verify the property values listed in Tables 1.5 and 1.6 by writing them with a script. This time we'll dispense with using "window" in the dot notation. When no particular window is specified, the interpreter assumes you are referring to the current window. While Tables 1.5 and 1.6 shows the formal dot notation to access those properties, we can simplify and shorten our code a little by letting the interpreter assume that the current window is the one whose document we want to access, like this:

```
document.MyForm.method
document.MyForm.action
document.MyForm.Visitor.name
document.MyForm.Visitor.type
document.MyForm.Visitor.value
```

peek ahead

You'll learn how to write content to pop-up windows in Chapter 11, "Windows and Frames."

This saves us some typing. Keep in mind, however, that sometimes it is necessary to reference a specific window object by name. For instance, if you wanted to write content to a pop-up window, you'd have to reference that pop-up window by name to call the `document.write` method for that window.

```
1   <html>
2   <head>
3   <title>Script 1.7: Examining Properties of the Form and Text
4        Element</title>
5   </head>
6   <body>
7   <img name="Cat" src="MyCat.jpg" height="100" width="166">
8   <form name="MyForm" method="post" action="mailto:sam@pooch.com">
9        Name: <input type="text" name="Visitor" value="Sam">
10  </form>
11  <script language="JavaScript" type="text/javascript">
12    document.write("<h2>Some Properties of the MyForm Form
13        Object</h2>")
14    document.write("Name: ", document.MyForm.name, "<br>")
15    document.write("Method: ", document.MyForm.method, "<br>")
16    document.write("Action: ", document.MyForm.action, "<br>")
```

```
17
18    document.write("<h2>Some Properties of the Visitor Text
19        Element</h2>")
20    document.write("Name: ", document.MyForm.Visitor.name, "<br>")
21    document.write("Type: ", document.MyForm.Visitor.type, "<br>")
22    document.write("Value: ", document.MyForm.Visitor.value,
23        "<br>")
24  </script>
25  </body>
26  </html>
```

Script 1.7   Examining Properties of the Form and Text Element

Save and view your file. The result should look similar to this:

Figure 1.20   Results of Script 1.7

OK, so this may not be the most useful script in the book in terms of use on a Web site. Still, it's good to know how to access and write the values of object properties. When we discuss **debugging**, we'll use a similar technique to periodically check the values of object properties. Debugging is the process of finding and eliminating coding errors. Now on to the event model.

The Event Model

While the Document Object Model supported by a particular browser specifies the objects and their associated properties and methods that we may manipulate and use, the **event model** specifies the browser and user events to which we, as programmers, may respond.

Events Are the "Active" in Interactive

Events are occurrences generated by the browser, such as loading a document, or by the user, such as moving the mouse. They are the user and browser activities to which we may respond dynamically with a scripting language like JavaScript. Here's a short list of events that Web developers like to capture:

Event	Description
Load	Browser finishes loading a Web document
Unload	Visitor requests a new document in the browser window
Mouseover	Visitor moves the mouse over some object in the document window
Mouseout	Visitor moves the mouse off of some object in the document window
Click	Visitor clicks the mouse button
Focus	Visitor gives focus to or makes active a particular window or form element by clicking on it with a mouse or other pointing device or tabbing to it
Blur	Visitor removes focus from a window or form element
Change	Visitor changes the data selected or contained in a form element
Submit	Visitor submits a form
Reset	Visitor resets a form

Table 1.7   *Most Popularly Captured Events*

There are several more events that we can capture with JavaScript, but the ones listed above are, by far, the most popular. We'll take an in-depth look at the complete event model in Chapter 5, "Events and Event Handlers."

Event Handlers Are the "Inter" in Interactive

While events are the activities we, as Web developers, can respond to, event handlers are the mechanisms that allow us to capture and actually respond to events with JavaScript statements. Event handlers make Web pages *inter*-active. Here are the event handlers that correspond to the above-listed events:

Event	Event Handler
Load	onLoad
Unload	onUnload
Mouseover	onMouseover
Mouseout	onMouseout
Click	onClick
Focus	onFocus
Blur	onBlur
Change	onChange
Submit	onSubmit
Reset	onReset

Table 1.8   Event Handlers Corresponding to Events Listed in Table 1.7

Event handlers are written inline with HTML like tag attributes. For instance, to display an alert box that says "Welcome" when a document loads, write the event handler as you would any other HTML tag attribute, complete with quotes around the value associated with the attribute:

```
<body onLoad="alert('Welcome!')">
```

Try it. Type the following in your favorite editor and save it as a text file with an .html extension.

```
1   <html>
2   <head>
3   <title>Script 1.8: An onLoad Event Handler</title>
4   </head>
5   <body onLoad="alert('Welcome!')">
6
7   </body>
8   </html>
```

Script 1.8   An onLoad Event Handler

Open the document in your Web browser. An alert box should pop up with the message "Welcome" that looks similar to this:

Figure 1.21   Results of Script 1.8

The title and exact appearance of an alert box vary by browser. Don't worry if yours doesn't look exactly like this. The message should be the same, though: "Welcome!"

Once the browser loaded the document, a *load* event occurred. The onLoad event handler in line 5 of Script 1.8 *handled* the event by generating an alert box with the message "Welcome!" Cool, huh? In later chapters, we'll examine how to capture other events and respond to them with JavaScript. For now, how about a recap?

side note

`alert` is a method of the `window` object. It takes a single string parameter representing the text you want to display in the alert box. Technically, its formal reference is `window.alert("Some Text")`. However, like with `document`, JavaScript assumes we want to use the current window, so most of the time we can leave off the window designation.

Summary

JavaScript is an object-based scripting language modeled after C++. As a *scripting language* it is easy to use, easy to learn, is loosely typed, and was created for the express purpose of adding interactivity to Web pages. JavaScript is not Java, nor is it a sub-language of Java. JavaScript was created by Brendan Eich of Netscape and was first made available in 1995 as part of the Netscape Navigator 2.0 browser.

JavaScript comes in three flavors: Core JavaScript, which includes the basic constructs of the language; Client-Side JavaScript, which adds access to browser objects and elements in a Web page; and Server-Side JavaScript, which allows interactivity with server-side databases and other resources. Because JavaScript is an object-based language, it makes extensive use of objects and their inherent properties and methods.

The key to HTML/JavaScript functionality is the Document Object Model. The Document Object Model exposes HTML and browser elements to JavaScript for manipulation. The DOM represents *possibilities*. An instance hierarchy, on the other hand, is a model of a particular Web document in memory. The instance hierarchy and references to the objects modeled within it are created in memory as the Web page loads.

While the Document Object Model supported by a particular browser specifies the Web page and browser objects and their associated properties and methods, the event model specifies the browser and user *events* to which we may respond. Events are the browser and user activities that occur. Event handlers allow us to define how to handle those events with JavaScript.

Review Questions:

1. Who developed JavaScript?
2. List three common features of scripting languages.
3. List and describe the three "flavors" of JavaScript.
4. Is JavaScript a sub-language of Java? Explain.
5. What's the difference between a compiled language and an interpreted language?
6. What advantages do interpreted languages offer?
7. What is a Document Object Model?
8. What is an instance hierarchy as described in this book? When is it created?
9. Who is the World Wide Web Consortium (W3C)?
10. What does the W3C have to do with JavaScript?
11. Who is the European Computer Manufacturer's Association (ECMA)?
12. What does the ECMA have to do with JavaScript?
13. Define each of the following:
    a. object
    b. property
    c. method
14. Where do JavaScript objects come from?
15. What determines which objects in a Web document are accessible in a particular browser?
16. When are objects actually created?
17. What was the first browser to provide support for JavaScript?
18. What was Netscape's original name for the DOM?
19. What's the difference between an event and an event handler?
20. When does a load event occur?

Challenge Questions

1. Why don't browser vendors adhere to the standards established by the World Wide Web Consortium? Or do they? Explain. (Hint: Read Appendix E, "Evolution of the Document Object Model.")

2. What object, added to the DOM with the release of Netscape Navigator 3, is one of most popular objects used in JavaScript? (Hint: Read Appendix E.)
3. What choices does a Web developer have when endeavoring to write code that works in most browsers? (Hint: Read Appendix E.)

Exercises

1. Visit Netscape's "JavaScript Documentation" Web site (http://developer.netscape.com/docs/manuals/javascript.html). Check out the latest JavaScript specification. Download the current "Client-Side JavaScript Reference" for use in your scripting. This reference is invaluable.
2. Visit Opera Software's Web site (http://www.opera.com/). Download and install the latest browser. Visit the support page and examine the latest developments and JavaScript support criteria.
3. Visit Netscape's Web site (http://www.netscape.com/). Download and install the latest browser.
4. Visit Microsoft Internet Explorer's Web site (http://www.microsoft.com/ie/). Download and install the latest browser.
5. Suppose you have an object named `Rose` whose color property was set to pink.
   a. Write the appropriate dot notation to access the `Rose` object's color property.
   b. Write the appropriate code to change the `Rose` object's color property to yellow.
6. Draw a diagram describing the instance hierarchy a browser would create associated with the following HTML code:

```
<html>
<head>
      <title>My Document</title>
</head>
<body bgcolor="yellow" text="black" link="blue" vlink="purple">
<img name="MailBanner" src="banner.gif" width="600" height="100">
<form name="MailingList" method="post">
      Name: <input type="text" name="Visitor" value="Devan"><br>
Email: <input type="text" name="Email" value="devan@kitcat.com"><br>
      <input type="submit" name="SubmitButton">
<input type="reset" name="ResetButton">
</form>
</body>
</html>
```

   a. Write the complete dot notation for each object and property in your diagram.
   b. Modify the Web document to write the values of the Visitor and Email fields at the end of the page.
   c. Add JavaScript statements to change the background color to purple and the text (foreground) color to yellow.

d. Using the JavaScript object reference in Appendix A, write the complete dot notation to call at least one method for each object in your diagram.

7. Visit the Web site for the W3C (http://www.w3.org). Check out the latest HTML specification.

8. Visit the Web site for the ECMA (http://www.ecma.ch). Check out the latest ECMA script specification.

9. Write the appropriate JavaScript statement to print your name in bold letters and your town in italics on the same line.

10. Write the appropriate JavaScript statement to print your name in bold letters and your town in italics on separate lines.

11. Write two JavaScript statements, one to print your name and one to print ", aka" and your nickname. Your nickname should be written in italics. Both your name and your nickname should appear in the document on the same line like this:

<p align="center">Tina Spain McDuffie, aka *WebWoman*</p>

12. Many Web sites use JavaScript to add interactivity to their Web pages and to create special effects. Search the Web for sites (personal, professional, or commercial) that you think use JavaScript. Describe the interactivity of a special effect you think was created with JavaScript. Explain whether you think the scripting was effective; that is, was it helpful, fun, useful, or just annoying. Would you use something similar on your Web site?

13. Search the Web for three articles about good Web design. Summarize what you found.

Scripting Exercises

Create a folder named assignment01 to hold the documents you create during this assignment.

1. Create a document named exploringJOR.html. Using the JavaScript object reference in Appendix A as a guide to the appropriate document properties, write a script to display the following information in a Web browser:
   a. The title of the Web document
   b. The location of the Web document
   c. The date the document was last modified

2. Create a personal home page named home.html with the following content:
   a. Your name written in the <title> and an <h1> tag at the top.
   b. A picture of yourself after or next to the heading. If you don't have a digital picture of yourself, use clip art or a favorite landscape.
   c. Add the following sections using the tags listed:
      i. <h2>My Assignments</h2>—leave space underneath for later input.
      ii. <h2>About Me</h2>—again leave space for later input.
      iii. <h2>Bio</h2>—complete this section with some biographical information in paragraphs <p> . . . </p>.

iv. <h2>My Hobbies</h2>—leave blank for later input.

v. <h2>My Work/Job</h2>—leave blank for later input.

vi. <h2>My Browser</h2>—leave blank for later input.

vii. <address>Contact Info</address>—leave blank for now.

viii. <p>Last Update: </p>—we'll add more later.

3. Create a home page named toyStore.html for a company that sells toys. Include images of balls, jacks, whistles, skateboards, dolls, and other items that the toy company sells. You should be able to find the images you need from an online clip art library.

4. Open the personal home page you created. Insert a script under the My Browser heading that prints the following information about your browser:

   a. appCodeName

   b. appName

   c. appVersion

   d. platform

   e. userAgent

5. Open the personal home page you created. Insert a script within the Last Update paragraph right after the words "Last Update:" to write the date when the document was last modified. (Hint: Examine the properties of the document object in Appendix A to find the appropriate property to supply this info.)

6. Open the personal home page you created. Under the image, add a script that writes the `src` property and dimensions of the image. (Hint: Give the image a name with HTML if you haven't already, so you can easily access its properties. Don't put any spaces in the image's `name` attribute.)

7. Open the personal home page you created. In the My Browser section, add a script that displays the current screen resolution, like the following example:

<div align="center">Screen Resolution: 800 x 600</div>

Of course, the numbers should be provided by the appropriate property. (Hint: Check out the screen object in Appendix A.)

8. Open the personal home page you created. In the My Browser section, add a script that displays the current window's dimensions, like this:

<div align="center">Window Dimensions: 452 x 300</div>

where the numbers are provided by the appropriate window properties. (Hint: Look up the `window` object in Appendix A. This may not work in all browsers; experiment with several different properties.)

9. Open the personal home page you created. In the My Browser section, add a script that displays the current document's location. Do not use the `href` property. Instead, write the protocol, host, and path name one after the other to display the document's URL.