Part I

Getting Started with JavaScript

IN THIS PART

Chapter 1
JavaScript's Role in the World Wide Web and Beyond

Chapter 2
Developing a Scripting Strategy

Chapter 3
Selecting and Using Your Tools

Chapter 4
JavaScript Essentials

CHAPTER

1

JavaScript's Role in the World Wide Web and Beyond

Many of the technologies that make the World Wide Web possible have far exceeded their original goals. Envisioned at the outset as a medium for publishing static text and image content across a network, the Web is forever being probed, pushed, and pulled by content authors. By taking for granted so much of the "dirty work" of conveying the bits between server and client computers, content developers and programmers dream of exploiting that connection to generate new user experiences and practical applications. It's not uncommon for a developer community to take ownership of a technology and mold it to do new and exciting things. But with so many web technologies — especially browser programming with JavaScript — within reach of everyday folks, we have witnessed an unprecedented explosion in turning the World Wide Web from a bland publishing medium into a highly interactive, operating system–agnostic authoring platform.

The JavaScript language, working in tandem with related browser features, is a web-enhancing technology. When employed on the client computer, the language can help turn a static page of content into an engaging, interactive, and intelligent experience. Applications can be as subtle as welcoming a site's visitor with the greeting "Good morning!" when it is morning in the client computer's time zone — even though it is dinnertime where the server is located. Or, applications can be much more obvious, such as delivering the content of a slide show in a one-page download while JavaScript controls the sequence of hiding, showing, and "flying slide" transitions as we navigate through the presentation.

Of course, JavaScript is not the only technology that can give life to drab web content. Therefore, it is important to understand where JavaScript fits within the array of standards, tools, and other technologies at your disposal. The alternative technologies described in this chapter are HTML, Cascading Style Sheets (CSS), server programs, and plug-ins. In most cases, JavaScript can work side by side with these other technologies, even though the hype can make them sound like one-stop shopping places for all your interactive needs. (That's rarely the case.) Finally, you learn about the origins of JavaScript and what role it plays in today's advanced web browsers.

IN THIS CHAPTER

How JavaScript blends with other web-authoring technologies

The history of JavaScript

What kinds of jobs you should and should not entrust to JavaScript

Competing for Web Traffic

Web page publishers revel in logging as many visits to their sites as possible. Regardless of the questionable accuracy of web page hit counts, a site consistently logging 10,000 dubious hits per week is clearly far more popular than one with 1,000 dubious hits per week. Even if the precise number is unknown, relative popularity is a valuable measure. Another useful number is how many links from outside pages lead to a site. A popular site will have many other sites pointing to it — a key to earning high visibility in web searches.

Encouraging people to visit a site frequently is the Holy Grail of web publishing. Competition for viewers is enormous. Not only is the Web like a 50 million–channel television, but also, the Web competes for viewers' attention with all kinds of computer-generated information. That includes anything that appears onscreen as interactive multimedia.

Users of entertainment programs, multimedia encyclopedias, and other colorful, engaging, and mouse-finger-numbing actions are accustomed to high-quality presentations. Frequently, these programs sport first-rate graphics, animation, live-action video, and synchronized sound. By contrast, the lowest-common-denominator web page has little in the way of razzle-dazzle. Even with the help of Dynamic HTML and style sheets, the layout of pictures and text is highly constrained compared with the kinds of desktop publishing documents you see all the time. Regardless of the quality of its content, an unscripted, vanilla HTML document is flat. At best, interaction is limited to whatever navigation the author offers in the way of hypertext links or forms whose filled-in content magically disappears into the web site's server.

Other Web Technologies

With so many ways to spice up web sites and pages, you can count on competitors for your site's visitors to do their darnedest to make their sites more engaging than yours. Unless you are the sole purveyor of information that is in high demand, you continually must devise ways to keep your visitors coming back and entice new ones. If you design for an intranet, your competition is the drive for improved productivity by colleagues who use the internal web sites for getting their jobs done.

These are all excellent reasons why you should care about using one or more web technologies to raise your pages above the noise. Let's look at the major technologies you should know about.

Figure 1-1 illustrates the components that make up a typical dynamic web site. The core is a dialog between the server (the web host) and the client (the browser); the client requests data and the server responds. The simplest model would consist of just the server, a document, and a browser, but in practice web sites use the other components shown here, and more.

The process begins when the browser requests a page from the server. The server delivers static HTML pages directly from its hard drive; dynamic pages are generated on-the-fly by scripts executed in a language interpreter, such as PHP, but likewise delivered by the server to the client, usually in the form of HTML markup. For example, a server-side database can store the items of a catalog, and a PHP script can look up those data records and mark them up as HTML when requested by the browser.

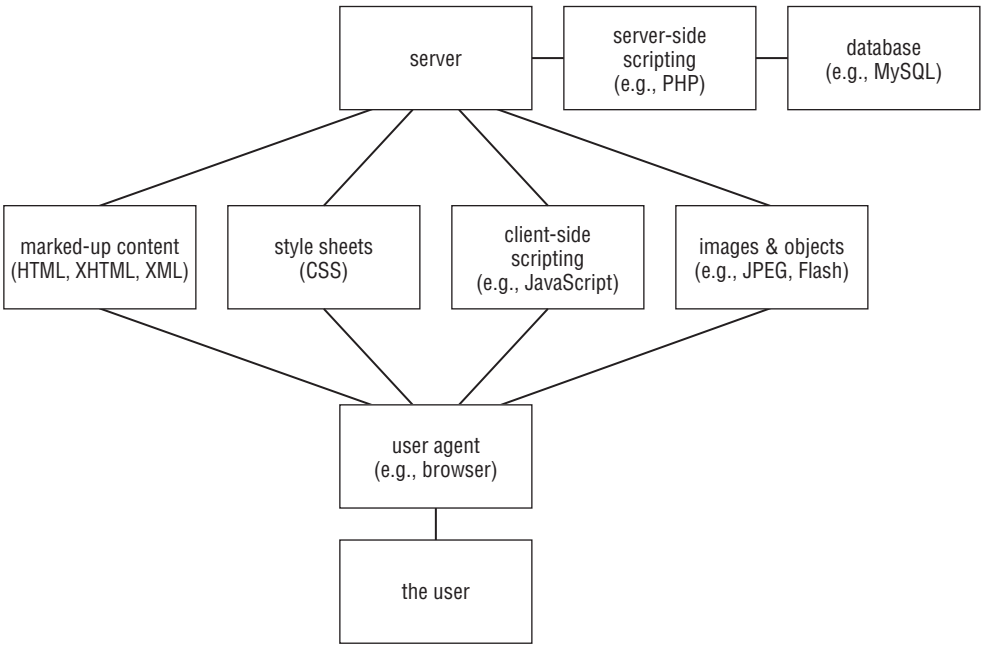
The downloaded page may contain the addresses of other components: style sheets, JavaScript files, images, and other assets. The browser requests each of these, in turn, from the server, combining them into the final rendering of the page.

A JavaScript program is inert — just another hunk of downloaded bits — until it's received by the browser, which validates its syntax and compiles it, ready for execution when called upon by the

HTML page or the human user. It's then part of the web page and confined to the client side of the sever-client dialog. JavaScript can make requests of servers, as we'll see later, but it cannot directly access anything outside of the page it's part of and the browser that's running it.

FIGURE 1-1

The components of a typical dynamic web site.



Hypertext Markup Language (HTML and XHTML)

As an outgrowth of SGML (Standard Generalized Markup Language), HTML brings structure to the content of a page. This structure gives us handles on the content in several important ways:

- Markup transforms a sea of undifferentiated text into discrete parts, such as headlines, paragraphs, lists, data tables, images, and input controls. Structure augments the meaning of the content by establishing relationships between different parts: between headline and subhead; between items in a list; between divisions of the page such as header, footer, and content columns. The semantics of a page are important to every reader: visual browsers for the sighted, vocal and Braille browsers for the visually impaired, search engines, and other software that parses the page, looking for a comprehensible structure.
- The browser transforms some marked-up structures into objects with particular behaviors. An image spills out rows of pixels in visually meaningful patterns. A form control accepts input from the user, while a button submits those form control values to the server. A hyperlink can load a new page into the browser window. These different types of objects would not be possible without some form of markup.

- The way a page is presented on screen, in Braille, or in speech, is determined by a style sheet that points to elements on the page and assigns appearance, emphasis, and other attributes. Every browser comes with a default style sheet that makes headlines, body text, hyperlinks, and form controls look a particular way. As web developers, we create our own style sheets to override the browser defaults and make our pages look the way we want. A style sheet tells the browser how to render the HTML page by referring to the document's markup elements and their attributes.
- Most importantly for the topic at hand, HTML markup gives JavaScript ways to locate and operate on portions of a page. A script can collect all the images in a gallery, or jump to a particular paragraph, because of the way the document is marked up.

Clearly, HTML markup and the way we handle it are critical to a document's structure and meaning, its presentation, and its successful interaction with JavaScript. While all the details of how best to use HTML are outside the scope of this book, it's clear that you'll want to hone your markup skills while you're learning JavaScript. Even the best script in the world can fail if launched on faulty, sloppy, or unplanned markup.

In the early years of web development, back before the turn of the century, in that medieval era now known as "the 90s," web designers didn't have much consciousness about markup semantics and accessibility. Pages were marked up in any way that would produce a desired visual presentation, relying on the default styling of the browser and without regard for the sensibility of the markup. A developer might choose an `h4` tag simply in order to produce a certain size and style of font, regardless of the document's outline structure, or use data table markup for non-tabular content, simply to force the page layout to align. Required spaces and faux spacer images were mashed into the real content merely to tweak the appearance of the page: how things *looked* took complete precedence over what things *meant*. Fortunately, today that perspective seems quaintly old-fashioned, but unfortunately, there are still thousands of people producing pages that way and millions of pages left over from that dark era, their finger bones clutching at our sleeves and urging us to join them in their morbid pastime. It is one goal of this book to encourage you to code like a modern, living mammal, and not like a fossil.

Relegating HTML to the category of a tagging language does disservice not only to the effort that goes into fashioning a first-rate web page, but also to the way users interact with the pages. To our way of thinking, any collection of commands and other syntax that directs the way users interact with digital information is *programming*. With HTML, a web-page author controls the user experience with the content just as the engineers who program Microsoft Excel craft the way users interact with spreadsheet content and functions.

Version 4.0 and later of the published HTML standards endeavor to define the purpose of HTML as assigning context to content, leaving the appearance to a separate standard for style sheets. In other words, it's not HTML's role to signify that some text is italic but rather to signify *why* we might choose to italicize it. For example, you would tag a chunk of text that conveys emphasis (via the `<em>` tag) or to mark its purpose (`<span class="title">`) separately from the decision of how to format it with the style sheet. This separation between HTML markup and CSS presentation is an extremely powerful concept that makes the tweaking and redesign of web sites much faster than in the old days of inline styling and `<font>` tags.

XHTML is a more recent adaptation of HTML that adheres to stylistic conventions established by the XML (eXtensible Markup Language) standard. No new tags come with XHTML, but it reinforces the notion of tagging to denote a document's structure and content. While for several years XHTML was seen as the next stage of evolution of HTML toward the holy grail of a universal XML markup for documents, that promise has failed to deliver, in part because Microsoft, with its enormous share of the browser market, has consistently declined to accommodate XHTML served correctly as `application/xhtml+xml`. Nearly all XHTML documents

today are served as `text/html`, which means that browsers treat them as just more HTML "tag soup." Perhaps the only advantage gained by using XHTML markup is the stricter set of rules used by the W3C HTML Validator (`http://validator.w3.org`), which checks to make sure that all tags are closed in XHTML documents, but isn't so fussy with HTML with its looser definitions.

For convincing arguments against the use of XHTML served as `text/html`, read "Sending XHTML as text/html Considered Harmful," by Ian Hickson (`http://hixie.ch/advocacy/xhtml`) and "Beware of XHTML," by David Hammond (`http://www.webdevout.net/articles/beware-of-xhtml`).

More recently, HTML5 has been growing in the Petri dishes of the World Wide Web Consortium (W3C). While modern browsers have only just begun to implement some of its features, HTML5 is considered by many to offer a more promising future than the seemingly abandoned XHTML. HTML5 offers an enriched vocabulary of markup tags compared to version 4, the better to align the markup language with the uses to which HTML is put in the real world.

Because HTML 4.01 is the prevailing markup language for the web today, the HTML samples in this book are compatible with HTML 4.01, except for those that illustrate newer HTML5 elements such as `canvas`. With the proper `DOCTYPE`, the samples should validate as HTML5. They are easily convertible to XHTML1.0 with a few simple transforms: change the `DOCTYPE`, add `lang` attributes to the `HTML` element, and close empty elements such as `input` and `link` with `/>`. It's our hope that this will make the samples useful to folks heading up either HTML or XHTML paths, and also make the book more future-friendly as HTML5 comes into its own.

Cascading Style Sheets (CSS)

Specifying the look and feel and speech of a web page is the job of Cascading Style Sheets (CSS). Given a document's structure as spelled out by its HTML markup, a style sheet defines the layout, colors, fonts, voices, and other visual and aural characteristics to present the content. Applying a different set of CSS definitions to the same document can make it look and sound entirely different, even though the words and images are the same.

(CSS 2.1 is the version of the W3C style sheet specification most widely supported by today's user agents. Aural style sheets that let us assign voices and other sounds to the markup of a web page are a module of the CSS 3 specification and, at this writing, are supported only by Opera and the FireVox extension to Firefox, although other browsers are sure to follow.)

Mastery of the fine points of CSS takes time and experimentation, but the results are worth the effort. The days of using HTML tables and transparent "spacer" images to generate elaborate multicolumn layouts are very much on the wane. Every web developer should have a solid grounding in CSS.

The learning curve for CSS can be steep because of the inconsistent support for its many features from one browser to the next. You can make your life much easier by triggering *standards mode* in the browser, whereby it adheres more closely to the W3C CSS specification. We recommend triggering standards mode by using a correct `DOCTYPE` at the top of your markup, such as one of these:

```
HTML 4.01 Strict:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/REC-html40/strict.dtd">

HTML 5:

<!DOCTYPE html>
```

```
XHTML 1.0 Strict:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/xhtml1-strict.dtd">
```

For a clear explanation of `DOCTYPE` switching, read Eric Meyer's essay "Picking a Rendering Mode" (`http://www.ericmeyeroncss.com/bonus/render-mode.html`).

Server-side programming

Web sites that rely on database access, or change their content very frequently, incorporate programming on the server that generates the HTML output for browsers and/or processes forms that site visitors fill out on the page. Even submissions from a simple login or search form ultimately trigger some server process that sends the results to your browser. Server programming takes on many guises, the names of which you may recognize from your surfing through web development sites. PHP, ASP, .NET, JSP, and ColdFusion are among the most popular. Associated programming languages include Perl, Python, Java, C++, C#, Visual Basic, and even server-side JavaScript in some environments.

Whatever language you use, the job definitely requires the web-page author to be in control of the server, including whatever *back-end* programs (such as databases) are needed to supply results or massage the information coming from the user. Even with the new, server-based web site design tools available, server scripting is often a task that a content-oriented HTML author will need to hand off to a more experienced programmer.

Client-side JavaScript is not a replacement for server-side scripting. Any web site that reacts dynamically to user actions or that saves data must be driven by server-side scripting, even if JavaScript is used in the browser to enhance the user's experience. The reasons are simple: First, JavaScript itself cannot write to files on the server. It can help the user make choices and prepare data for upload, but after that it can only hand off data to a sever-side script for database updating. Second, not all user agents run JavaScript. Screen readers, mobile devices, search engines, and browsers installed in certain corporate contexts are among those that don't invoke JavaScript, or that receive web pages with the JavaScript stripped out. Therefore, your web site should be fully functional with JavaScript turned off. Use JavaScript to make the browsing experience faster, cooler, or more fun when it's present, but don't let your site be broken if JavaScript isn't running.

As powerful and useful as server-side scripting can be, its speed of interaction with the user is limited by the speed of the Internet connection between server and user agent. Obviously, any process that results in updating data on the server must include some client-server dialog, but there are many aspects of user interaction that, with the help of JavaScript, can take place entirely within the browser — form validation and drag-&-drop are two examples — then update the server when the response-sensitive process is complete.

One way that server programming and browser scripting work together is with what has become known as *Ajax* — Asynchronous JavaScript and XML. The "asynchronous" part runs in the browser, requesting XML data from, or posting data to, the server-side script entirely in the background. XML data returned by the server can then be examined by JavaScript in the browser to update portions of the web page. That's how many popular web-based email user interfaces work, as well as the draggable satellite-photo closeups of Google Maps (`http://maps.google.com`).

Working together, server programming and browser scripting can make beautiful applications together. You'll want to write in a server-side language such as PHP, or team up with someone who does, to lay the foundations for the JavaScript-enhanced pages you'll be creating.

Of helpers and plug-ins

In the early days of the World Wide Web, a browser needed to present only a few kinds of data before a user's eyes. The power to render text (tagged with HTML) and images (in popular formats such as GIF and JPEG) was built into browsers intended for desktop operating systems. Not wanting to be limited by those data types, developers worked hard to extend browsers so that data in other formats could be rendered on the client computer. It was unlikely, however, that a browser would ever be built that could download and render, say, any of several sound-file formats.

One way to solve the problem was to allow the browser, upon recognizing an incoming file of a particular type, to launch a separate application on the client machine to render the content. As long as this helper application was installed on the client computer (and the association with the helper program was set in the browser's preferences), the browser would launch the program and send the incoming file to that program. Thus, you might have one helper application for a MIDI sound file and another for an animation file.

Beginning with Netscape Navigator 2 in early 1996, software *plug-ins* for browsers enabled developers to extend the capabilities of the browser without having to modify the browser. Unlike a helper application, a plug-in can enable external content to blend into the document seamlessly.

The most common plug-ins are those that facilitate the playback of audio and video from the server. Audio may include music tracks that play in the background while visiting a page, or live (streaming) audio, similar to a radio station. Video and animation can operate in a space on the page when played through a plug-in that knows how to process such data.

Today's browsers tend to ship with plug-ins that decode the most common sound-file types. Developers of plug-ins for Internet Explorer for the Windows operating system commonly implement plug-ins as ActiveX controls — a distinction that is important to the underpinnings of the operating system, but not to the user.

Plug-ins and helpers are valuable for more than just audio and video playback. A popular helper application is *Adobe Acrobat Reader,* which displays Acrobat files that are formatted just as though they were being printed. But for interactivity, developers today frequently rely on the *Flash* plug-in by Macromedia (now owned by Adobe). Created using the Flash authoring environment, a Flash application can have active clickable areas and draggable elements, animation, and embedded video. Some authors simulate artistic video games and animated stories in Flash. A browser equipped with the Flash plug-in displays the content in a rectangular area embedded within the browser page. A variant of JavaScript called *ActionScript* enables Flash to interact with the user and components of the HTML page. Like JavaScript, ActionScript gets access to outside resources by making data-read and -write requests of the server. YouTube.com is a popular example of a web site with richly integrated Flash.

One potential downside for authoring interactive content in Flash or similar environments is that if the user does not have the correct plug-in version installed, it can take some time to download the plug-in (if the user even wants to bother). Moreover, once the plug-in is installed, highly graphic and interactive content can take longer to download to the client (especially on a dial-up connection) than some users are willing to wait. This is one of those situations in which you must balance your creative palette with the user's desire for your interactive content.

Another client-side technology — the Java applet — was popular for a while in the late 1990s, but has fallen out of favor for a variety of reasons (some technical, some corporate-political). But this has not diminished the use of Java as a language for server and even cellular telephone programming, extending well beyond the scope of the language's founding company, Sun Microsystems.

JavaScript: A Language for All

Sun's Java language is derived from C and C++, but it is a distinct language. Its main audience is the experienced programmer. That leaves out many web-page authors. Java's preliminary specifications in 1995 were dismaying. How much more preferable would have been a language that casual programmers and scripters who were comfortable with authoring tools such as Apple's once-formidable HyperCard and Microsoft's Visual Basic could adopt quickly. As these accessible development platforms have shown, nonprofessional authors can dream up many creative applications, often for very specific tasks that no professional programmer would have the inclination to work on. Personal needs often drive development in the classroom, office, den, or garage. But Java was not going to be that kind of inclusive language.

Spirits lifted several months later, in November 1995, when we heard of a scripting language project brewing at Netscape Communications, Inc. Born under the name *LiveScript,* this language was developed in parallel with a new version of Netscape's web server software. The language was to serve two purposes with the same syntax. One purpose was as a scripting language that web server administrators could use to manage the server and connect its pages to other services, such as back-end databases and search engines for users looking up information. Extending the "Live" brand name further, Netscape assigned the name *LiveWire* to the database connectivity usage of LiveScript on the server.

On the client side — in HTML documents — authors could employ scripts written in this new language to enhance web pages in a number of ways. For example, an author could use LiveScript to make sure that the user had filled in a required text field with an e-mail address or credit card number. Instead of forcing the server or database to do the data validation (requiring data exchanges between the client browser and the server), the user's computer handles all the calculation work — putting some of that otherwise-wasted computing horsepower to work. In essence, LiveScript could provide HTML-level interaction for the user.

LiveScript becomes JavaScript

In early December 1995, just prior to the formal release of Navigator 2, Netscape and Sun Microsystems jointly announced that the scripting language thereafter would be known as JavaScript. Though Netscape had several good marketing reasons for adopting this name, the changeover may have contributed more confusion to both the Java programming world and HTML scripting world than anyone expected.

Before the announcement, the language was already related to Java in some ways. Many of the basic syntax elements of the scripting language were reminiscent of the Java style. However, for client-side scripting, the language was intended for very different purposes than Java — essentially to function as a programming language integrated into HTML documents rather than as a language for writing applets that occupy a fixed rectangular area on the page (and that are oblivious to anything else on the page). Instead of Java's full-blown programming language vocabulary (and conceptually more difficult-to-learn object-oriented approach), JavaScript had a small vocabulary and a more easily digestible programming model.

The true difficulty, it turned out, was making the distinction between Java and JavaScript clear to the world. Many computer journalists made major blunders when they said or implied that JavaScript provided a simpler way of building Java applets. To this day, some new programmers believe JavaScript is synonymous with the Java language: They post Java queries to JavaScript-specific Internet newsgroups and mailing lists.

The fact remains that client-side Java and JavaScript are more different than they are similar. The two languages employ entirely different interpreter engines to execute their code.

Enter Microsoft and others

Although the JavaScript language originated at Netscape, Microsoft acknowledged the potential power and popularity of the language by implementing it (under the JScript name) in Internet Explorer 3. Even if Microsoft might prefer that the world use the VBScript (Visual Basic Script) language that it provides in the Windows versions of IE, the fact that JavaScript is available on more browsers and operating systems makes it the client-side scripter's choice for anyone who must design for a broad range of users.

With scripting firmly entrenched in the mainstream browsers from Microsoft and Netscape, newer browser makers automatically provided support for JavaScript. Therefore, you can count on fundamental scripting services in browsers such as Opera or the Apple Safari browser (the latter built upon an Open Source browser called KHTML). Not that all browsers work the same way in every detail — a significant challenge for client-side scripting that is addressed throughout this book.

JavaScript versions

The JavaScript language has its own numbering system, which is completely independent of the version numbers assigned to browsers. The Mozilla Foundation, successor to the Netscape browser development group that created the language, continues its role as the driving force behind the JavaScript version numbering system.

The first version, logically enough, was JavaScript 1.0. This was the version implemented in Navigator 2, and the first release of Internet Explorer 3. As the language evolved with succeeding browser versions, the JavaScript version number incremented in small steps. JavaScript 1.2 is the version that has been the most long lived and stable, currently supported by Internet Explorer 7. Mozilla-based browsers and others have inched forward with some new features in JavaScript 1.5 (Mozilla 1.0 and Safari), JavaScript 1.6 (Mozilla 1.8 browsers), and JavaScript 1.7 (Mozilla 1.8.1 and later).

Each successive generation of JavaScript employs additional language features. For example, in JavaScript 1.0, arrays were not developed fully, causing scripted arrays not to track the number of items in the array. JavaScript 1.1 filled that hole by providing a constructor function for generating arrays and an inherent `length` property for any generated array.

The JavaScript version implemented in a browser is not always a good predictor of core language features available for that browser. For example, although JavaScript 1.2 (as implemented by Netscape in Netscape Navigator 4) included broad support for regular expressions, not all of those features appeared in Microsoft's corresponding JScript implementation in Internet Explorer 4. By the same token, Microsoft implemented `try-catch` error handling in its JScript in Internet Explorer 5, but Netscape didn't include that feature until the Mozilla-based Netscape Navigator 6 implementation of JavaScript 1.5. Therefore, the language version number is an unreliable predictor in determining which language features are available for you to use.

Core language standard: ECMAScript

Although Netscape first developed the JavaScript language, Microsoft incorporated the language in Internet Explorer 3. Microsoft did not want to license the Java name from its trademark owner (Sun Microsystems), which is why the language became known in the Internet Explorer environment as

JScript. Except for some very esoteric exceptions and the pace of newly introduced features, the two languages are essentially identical. The levels of compatibility between browser brands for a comparable generation are remarkably high for the core language (unlike the vast disparities in object model implementations discussed in Chapter 25, "Document Object Model Essentials").

As mentioned, standards efforts have been under way to create industry-wide recommendations for browser makers to follow (to make developers' lives easier). The core language was among the first components to achieve standard status. Through the European standards body called ECMA, a formal standard for the language was agreed to and published. The first specification for the language, dubbed ECMAScript by the standards group, was roughly the same as JavaScript 1.1 in Netscape Navigator 3. The standard (ECMA-262) defines how various data types are treated, how operators work, what a particular data-specific syntax looks like, and other language characteristics. A newer version (called version 3) added many enhancements to the core language (version 2 was just version 1 with errata fixed).

The current version of the ECMAScript specification is known as ECMAScript, Fifth Edition, published online at `www.ecma-international.org`. To quote the ECMA, "The Fifth Edition codifies de facto interpretations of the language specification that have become common among browser implementations and adds support for new features that have emerged since the publication of the Third Edition. Such features include accessor properties, reflective creation and inspection of objects, program control of property attributes, additional array manipulation functions, support for the JSON object encoding format, and a strict mode that provides enhanced error checking and program security."

If you are a student of programming languages, you will find the document fascinating; if you simply want to script your pages, you might find the minutia mind-boggling.

All mainstream browser developers have pledged to make their browsers compliant with the ECMA standard. The vast majority of the ECMAScript standard has appeared in Navigator since version 3 and Internet Explorer since version 4, and as new features are added to the ECMA standard, they tend to find their way into newer browsers as well. The latest version of ECMAScript is version 5. The previous edition, the 3rd, has been supported in all mainstream browsers for the past few years.

Note

Even as ECMAScript, Fifth Edition was in the works, The Mozilla Foundation and Microsoft were implementing comparable versions of JavaScript 2.0 and JScript, respectively. An extension to ECMAScript called E4X (ECMAScript for XML) was finalized in late 2005 and is implemented in browsers based on Mozilla 1.8.1 or later (for example, Firefox 2.0). The Adobe ActionScript 3 language, which is used in the development of Flash animations, fully supports E4X. E4X is a significant addition to JavaScript because it makes XML (Extensible Markup Language) a native data type within the syntax of the language, making the processing of data in XML format much easier. XML is the data format used by many data exchange processes, including Ajax (see Chapter 39). ■

JavaScript: The Right Tool for the Right Job

Knowing how to match an authoring tool to a solution-building task is an important part of being a well-rounded web site author. A web designer who ignores JavaScript is akin to a plumber who bruises his knuckles by using pliers instead of the wrench from the bottom of the toolbox.

By the same token, JavaScript won't fulfill every dream. The more you understand about JavaScript's intentions and limitations, the more likely you will be to turn to it immediately when it is the proper tool. In particular, look to JavaScript for the following kinds of solutions:

- Getting your web page to respond or react directly to user interaction with form elements (input fields, text areas, buttons, radio buttons, checkboxes, selection lists) and hypertext links
- Distributing small collections of database-like information and providing a friendly interface to that data
- Controlling multiple-frame navigation, plug-ins, or Java applets based on user choices in the HTML document
- Preprocessing data on the client before submission to a server
- Changing content and styles in modern browsers dynamically and instantly, in response to user interaction
- Requesting files from the server, and making read and write requests of server-side scripts

At the same time, it is equally important to understand what JavaScript is *not* capable of doing. Scripters waste many hours looking for ways of carrying out tasks for which JavaScript was not designed. Most of the limitations are intentional, to protect visitors from invasions of privacy or unauthorized access to their desktop computers. Therefore, unless a visitor uses a modern browser and explicitly gives you permission to access protected parts of his or her computer, JavaScript cannot surreptitiously perform any of the following actions:

- Setting or retrieving the browser's preferences settings, main window appearance features, action buttons, and printing capability
- Launching an application on the client computer
- Reading or writing files or directories on the client computer (with one exception: cookies)
- Writing directly to files on the server
- Capturing live data streams from the server for retransmission
- Sending secret e-mails from web site visitors to you (although it can send data to a server-side script capable of sending email)

Web site authors are constantly seeking tools that will make their sites engaging (if not cool) with the least amount of effort. This is particularly true when the task of creating a web site is in the hands of people more comfortable with writing, graphic design, and page layout than with hard-core programming. Not every webmaster has legions of experienced programmers on hand to whip up some special, custom enhancement for the site. Neither does every web author have control over the web server that physically houses the collection of HTML and graphics files. JavaScript brings programming power within reach of anyone familiar with HTML, even when the server is a black box at the other end of a telephone line.