CHAPTER 2

Authoring Challenges amid the Browser Wars

In This Chapter

How leapfrogging browser developments hurt Web developers

Why the core JavaScript language is separate from the document objects

How to develop a cross-platform strategy

If you are learning JavaScript just now in the brief history of scriptable browsers, you have both a distinct advantage and disadvantage. The advantage is that you have the wonderful capabilities of the level 4 browser offerings from Netscape and Microsoft at your bidding. The disadvantage is that you have not experienced the painful history of authoring for older browser versions that were buggy and at times incompatible with each other due to a lack of standards. You have yet to learn the anguish of carefully devising a scripted application for the browser version you use, only to have site visitors sending you voluminous e-mail messages about how the page triggers all kinds of script errors when run on a different browser brand, generation, or operating system platform.

Welcome to the real world of scripting Web pages in JavaScript. Several dynamics are at work to help make an author's life difficult if the audience for the application uses more than a single type of browser. This chapter introduces you to these challenges before you type your first word of JavaScript code. My fear is that the subjects I raise may dissuade you from progressing further into JavaScript and its powers. But as a developer myself — and as someone who has been using JavaScript since the earliest days of its public prerelease availability — I dare not sugarcoat the issues facing scripters today. Instead, I want to make sure you have an appreciation of what lies ahead to assist you in learning the language. I believe if you understand the Big Picture of the browser scripting world as it stands near the beginning of 1998, you will find it easier to target JavaScript usage in your Web application development.

Leapfrog

Browser compatibility has been an issue for authors since the earliest days of rushing to the Web. Despite the fact that browser developers and other interested parties had their voices heard during formative stages of standards development, HTML authoring has rarely been the “write once, run everywhere” phenomenon that Sun Microsystems promises for the Java development environment. It may have been one thing to establish a set of standard tags for defining heading levels and line breaks, but it was rare for the actual rendering of content inside those tags to look identical on two different brands of browsers.

Then, as the competitive world heated up — and Web browser developers transformed themselves from volunteer undertakings into profit-seeking businesses — creative people defined new features and new tags that helped authors create more flexible and interesting-looking pages. As happens a lot in any industry that is computer-related, the pace of commercial development easily outpaced the studied processing of standards. A browser maker would build a new HTML feature into a browser and propose that feature to the relevant standards body. Web authors were using these features (sometimes for prerelease browser versions) before the proposals were published for review.

When the deployment of content depends almost entirely on an interpretive engine on the client computer receiving the data — the HTML engine in a browser, for example — authors face an immediate problem. Unlike a standalone computer program that can extend and even invent functionality across a wide range and have it run on everyone’s computer (at least for a given operating system), Web content providers must rely on the functionality built into the browser. If not all browsers coming to my site support a particular HTML feature, then should I apply newfangled HTML features for visitors only at the bleeding edge? And if I do deploy the new features, what do I do for those with older browsers?

Authors who developed pages in the earliest days of the Web wrestled with this question for many HTML features that we today take for granted. Tables and frames come to mind. Eventually the standards caught up with the proposed HTML extensions, but not without a lot of author anguish along the way.

The same game continues today. But the field of players has shrunk to two primary players: Netscape and Microsoft. And for these companies, the stakes are higher than ever before — market share, investor return on investment — pick a business buzzword, and you’ll find a reason behind the competition. What had begun years ago as a friendly game of leapfrog (long before Microsoft even acknowledged the Web) has become an out-and-out war: the Browser War.

Ducking for Cover

Sometimes it is difficult to tell from week to week where the battles are being fought. Marketing messages from the combatants turn on a dime. You can’t tell if the message is proactive to stress a genuinely new corporate strategy or reactive to match the opponent’s latest salvo. The combatants keep touting to each other: “Anything you can do, we can do better.”

If it were a case of Netscape and Microsoft pitching their server and browser software to customers for the creation of monolithic intranets, I could understand

and appreciate such efforts. The battle lines would be clearly drawn, and potential customers could base their decisions on unemotional criteria — how well the solution fits the customer's information distribution and connectivity goals. In fact, if your development environment is monolithic, you are in luck, because authoring for a single browser brand and minimum version is a piece of cake. But you are not in the majority.

As happens in war, civilian casualties mount when the big guns start shooting. The guns are going off around the clock with the release of level 4 browsers: Netscape Navigator 4 and Microsoft Internet Explorer 4. While a fair amount of common ground exists between the two browsers that authors can work with, the newest features cause the biggest problems for authors wishing to deploy on both browsers. Trying to determine where the common denominator is may be the toughest part of the authoring job.

Compatibility Issues Today

Allow me to describe the current status of compatibility between Navigator and Internet Explorer. The discussion in the next few sections intentionally does not get into specific scripting technology very deeply — some of you may know very little about programming. In many chapters throughout Parts III and IV, I offer scripting suggestions to accommodate both browsers.

Separating language from objects

Although early JavaScript authors initially treated client-side scripting as one environment that permitted the programming of page elements, the scene has changed a bit as the browsers have matured. Today, a clear distinction exists between specifications for the core JavaScript language and for the elements you script in a document (for example, buttons and fields in a form).

On one level, this separation is a good thing. It means that one specification exists for basic programming concepts and syntax that enables the same language to be applied to environments that may not even exist today. You can think of the core language as basic wiring. Once you know how electric wires work, you can connect them to all kinds of electrical devices, including some that may not yet be invented. Similarly, JavaScript today is used to wire together page elements in an HTML document. Tomorrow, the core language could be used by operating systems to let users wire together desktop applications that need to exchange information automatically.

At the ends of today's JavaScript wires are the elements on the page. In programming jargon, these items are known as *document objects*. By keeping the specifications for document objects separate from the wires that connect them, other kinds of wires (other languages) can be used to connect them. It's like designing telephones that can work with any kind of wire, including a type of wire that hasn't yet been invented. Today the devices can work with copper wire or fiber optic cable. You get a good picture of this separation in Internet Explorer, whose set of document objects can be scripted with JavaScript or VBScript. They're the same objects, just different wiring.

The separation of core language from document objects enables each concept to have its own standards effort and development pace. But even with recommended standards for each factor, each browser maker is free to extend the standards, and authors may have to expend more effort to devise one version of a page or script that plays on both browsers unless the script adheres to a common denominator (or uses some other branching techniques to let each browser run its own way).

Core language standard

Keeping track of JavaScript language versions requires study of history and politics. The history covers the three versions developed by Netscape; politics covers Microsoft's versions and the joint standards effort. The first version of JavaScript (in Navigator 2) was Version 1.0, although that numbering was not part of the language usage. JavaScript was JavaScript. Version numbering became an issue when Navigator 3 came out. The version of JavaScript associated with that Navigator version was JavaScript 1.1. As you will learn later in this book, the version number is sometimes necessary in an attribute of the HTML tags that surround a script. Navigator 4 increased the language version one more notch, with JavaScript 1.2.

Microsoft's scripting effort contributes confusion for scripting newcomers. The first version of Internet Explorer to include scripting was Internet Explorer 3. The timing of Internet Explorer 3 was roughly coincidental to Navigator 3. But as scripters soon discovered, Microsoft's scripting effort was one generation behind. Microsoft apparently did not (and may still not) have a license to the JavaScript name. As a result, the company called its language JScript. Even so, the HTML tag attribute that requires naming the language of the script inside the tags could be either JScript or JavaScript for Internet Explorer. A JavaScript script written for Navigator 2 would be understood by Internet Explorer 3.

During this period of dominance by Navigator 3 and Internet Explorer 3, scripting newcomers were often confused because they expected the scripting languages to be the same. Unfortunately for the scripters, there were language features in JavaScript 1.1 that were not available in the older JavaScript version in Internet Explorer 3. The situation smoothes out for Internet Explorer 4. Its core language is up to the level of JavaScript 1.2 in Navigator 4. Microsoft still officially calls the language JScript. Language features new in Navigator 4 (including the script tag attribute identifying JavaScript 1.2) are understood when the scripts load into Internet Explorer 4.

While all of this jockeying for JavaScript versions was happening, Netscape, Microsoft, and other concerned parties met to establish a core language standard. The standards body is a Switzerland-based organization called the European Computer Manufacturer's Association, or ECMA (commonly pronounced ECK-ma). In mid-1997, the first formal language specification was agreed on and published (ECMA-262). Due to licensing issues with the JavaScript name, the body created a new name for the language: ECMAScript.

With only minor and esoteric differences, this first version of ECMAScript is essentially the same as JavaScript 1.1 found in Navigator 3. Both Navigator 4 and Internet Explorer 4 support the ECMAScript standard. Moreover, as happens so often when commerce meets standards bodies, both browsers have gone beyond the ECMAScript standard. Fortunately, the common denominator of this extended core language is broad, lessening authoring headaches on this front.

Document Object Model

If Navigator 4 and Internet Explorer 4 are close in core JavaScript language compatibility, nothing could be further from the truth when it comes to the document objects. Internet Explorer 3 based its document object model (DOM) on that of Netscape Navigator 2, the same browser level it used as a model for the core language. When Netscape added a couple of new objects to the model in Navigator 3, the addition caused further headaches for neophyte scripters who expected those objects to be in Internet Explorer 3. Probably the most commonly missed object in Internet Explorer 3 was the image object, which lets scripts swap the image when a user rolls the cursor atop a graphic — *mouse rollovers*, they're commonly called.

In the level 4 browsers, however, Internet Explorer's document object model has jumped way ahead of the object model Netscape implemented in Navigator 4. You will see the benefits of this expanded object model in chapters that cover scripting Dynamic HTML (Chapters 41 through 43). Suffice it to say that it is an enviable implementation.

At the same time, a document object model standard is being negotiated under the auspices of the World Wide Web Consortium (W3C). The specification was in an incomplete draft stage in late 1997. An HTML-specific subset of the DOM standard will support object syntax implemented in Navigator 3. If you wish to script to a common denominator, most of the document object model in Navigator 4 is supported in Internet Explorer 4. Even so, scripting for objects that have been defined later than Navigator 3 is touchy business, and requires a good knowledge of compatibility, as described in the object discussions throughout this book.

Cascading Style Sheets

Navigator 4 and Internet Explorer 4 claim compatibility with a W3C recommendation called Cascading Style Sheets, Level 1 (CSS1). This specification for a way to customize content in an organized fashion throughout a document (and thus minimize the HTML in each tag) is an effort to extend the Web's tradition of publishing static content. A Level 2 version is currently in working draft form.

Neither company's implementation is 100 percent compatible with the W3C standard, although Internet Explorer 4 has fewer omissions than Navigator 4. Moreover, each browser has its own mutually incompatible extensions. Netscape's extensions are a completely different way of defining style sheets, using JavaScript syntax, rather than the recommended standard syntax. Internet Explorer 4 does not recognize these so-called JavaScript style sheets.

JavaScript also comes into play on the Internet Explorer side, because its object model embraces the implementation of CSS1 in the browser. Style sheet information is part of the object model, and is therefore accessible and modifiable from JavaScript. This is not the case in Navigator 4.

Dynamic HTML

Perhaps the biggest improvements to the inner workings of the level 4 browsers from both Netscape and Microsoft revolve around a concept called *Dynamic HTML* (DHTML). The ultimate goal of DHTML is to enable scripts in documents to control the content, content position, and content appearance in response to user actions.

To that end, the W3C organization is developing another standard for the precise positioning of HTML elements on a page as an extension of the CSS standards effort. The CSS-Positioning recommendation adds to the CSS1 syntax for specifying an exact location on the page where an element is to appear, whether the item should be visible, and what order it should take among all the items that might overlap it.

Internet Explorer 4 adheres to the CSS-P standard, and makes positionable items subject to script control. Navigator 4 follows the standard and implements an alternative methodology involving an entirely new, albeit unsanctioned, tag for layers. Such positionable items are scriptable in Navigator 4 as well, although some of the script syntax is different from that used in Internet Explorer 4. As a result, three chapters later in this book discuss the individual browser implementations of CSS-P as well as a way to script both versions in one page. This kind of cross-platform scripting can be challenging, yet is certainly possible if you understand the limitations imposed by following a common denominator.

Developing a Scripting Strategy

As you have seen in this chapter, the two level 4 browsers contain a hodgepodge of standards and proprietary extensions. Even if you try to script to a common denominator, it probably won’t take into account the earlier versions of both the JavaScript core language and the browser document object models.

The true challenge for authors these days is determining the audience for which scripted pages are intended. You will learn techniques in Chapter 13 that let you redirect users to different paths in your Web site based on their browser capabilities. Each new generation of browser not only brings with it new and exciting features you are probably eager to employ in your pages, it also adds to the fragmentation of the audience visiting a publicly accessible page. With each new upgrade, fewer existing users are ready to download tens of megabytes of browser merely to have the latest and greatest browser version. For many pioneers — and certainly for most nontechie users — there is an increasingly smaller imperative to upgrade browsers, unless that browser comes via a new computer or operating system upgrade.

As you work your way through the early parts of this book, know that many common denominators depend on where you wish to draw the line for browser support. Even if you wish to adhere to the absolutely lowest common denominator of scripting, I’ve got you covered: The tutorial of Part II focuses on language and object aspects that are compatible with every version of JavaScript.

At the same time, I think it is important for you to understand that the cool application you see running on your level 4 browser may not translate to Navigator 2. Therefore, when you see a technique that you’d like to emulate, be realistic in your expectations of adapting that trick for your widest audience. Only a good working knowledge of JavaScript and an examination of the cool source code will reveal how well it will work for your visitors.

✥ ✥ ✥