1

Introduction to JavaScript

Like many technologies that have enjoyed success and sticking power, JavaScript has taken on new purpose and relevance since its creation many years ago. It's no longer correct to say that JavaScript is *just* a scripting language or even *just* for the web. In fact, JavaScript is one of the few truly multi-vendor, multi-platform, and multi-purpose programming languages in use today. It holds this status not just because it happened to be the language that was designed for browser scripting but also because it's an extremely flexible, expressive, and forgiving language that both amateurs and professional developers alike can appreciate. Certainly one could say it's thanks to the web that we have such an interesting and powerful way to build applications, but it's thanks to JavaScript that we have such an interesting and powerful web.

This book will serve as a detailed reference for all things JavaScript. This includes, of course, all the language basics but also virtually everything to do with its core objects, features, and limitations. You'll examine advanced topics too, such as how JavaScript can be applied to provide specific interactivity or features inside a web page, how to use it to manipulate the structure of web documents, and how to interact with other web technologies like Flash, Silverlight, CSS, and even offline storage.

This chapter will provide an overview of the language and how it fits into the spectrum of web technologies. It'll provide some insight as to how someone typically learns the language and will explain both the history and current role of JavaScript amidst the cloud of competing browsers and interpreters. Finally, I'll introduce a simple web-based application using JavaScript and explain how it all fits together.

JavaScript History

Beginning its life as a decidedly curious enhancement to Netscape called *Mocha* (an homage to Java), JavaScript was intended for sparing use to add minor enhancements to the behavior of web pages, primarily to web forms. Netscape and Sun Microsystems evidently believed that this new dimensionality of the web could not, or should not, be addressed in the already complex declarative syntax of HTML. Instead, a scripting language was born that would continue to breathe life into the Internet for over a decade.

Before its full release, the name was changed to *LiveScript* and later to *JavaScript*. In March 1996, Netscape 2.0 was released to the world with the first official version the language. By August of the same year, Microsoft had released Internet Explorer 3.0 with a similar feature called JScript (but with some minor improvements). Over the coming years, the two companies would move virtually neck and neck with enhancements to the language. By June 1997, the international standards body Ecma approved a submission by Netscape to standardize the language. This standardized version of the language would be known as ECMAScript (ECMA-262) and was revised four times between 1997 and 2009. To this day there is some confusion in the developer community as to how "JavaScript," "ECMAScript," and "JScript" differ. The simple answer is that ECMAScript refers to the published and standardized version of the language, and JavaScript and JScript are dialects (or implementations) of that standard. Still, like other genericized brands that came before it such as Kleenex, Frisbee, Q-Tip, and Band-Aid, the JavaScript name stuck and probably always will.

Few programming languages have been as misunderstood as JavaScript. The root cause probably begins with the misleading name. JavaScript has very little to do with Java the language or even the Java Applet, another popular platform for web development in the early days. Compared to either, JavaScript was much smaller, simpler, and more purpose-built. Developers who wanted to use some of the more powerful features of Java in their JavaScript applications, such as class-based inheritance, were quickly confronted with these differences. In the years that followed the initial launch, the mood toward browser scripting swung wildly from enthusiasm to total distrust but recently to a place of high regard and rapid adoption. In the intervening years, a lot was added to the core language, and implicit cooperation between warring vendors like Microsoft and Mozilla allowed developers to write more to a single standard for the most part, irrespective of which browser people might be using.

Of course, JavaScript is no longer limited for use inside web pages. After being Ecma standardized, it was implemented as the scripting language of many other technologies like Flash, Adobe Acrobat, Microsoft .NET, and even as a way to write desktop widgets. In fact, the Ecma standard has undergone so many revisions in such a short period that the browser vendors no longer try to keep up, nor do they all agree what direction it's headed. While there are now many flavors of the language in a number of contexts, this book approaches the language from a position of practical use primarily as a way to script web pages. We have documented, as thoroughly as possible, where the core language leaves off and uniqueness of the browser picks up. In the appendices at the end of this book, you will find a detailed reference of the core language, how this maps to the Ecma standard, and also many of the browser extensions that have been added by individual vendors like Microsoft and Mozilla.

Looking Ahead to ES5 and Harmony

In 2007, the atmosphere of cooperation and collaboration among the biggest players in the browser space (Mozilla, Microsoft, Adobe, Google, and Opera) began to erode as architects put forth a proposal for ES4 (Edition 4) of the ECMAScript standard. This would be the most dramatic update to the standard in its history, introducing some radical features such as class-based inheritance, namespaces, and iterators. Participants from Microsoft strongly opposed both the nature of the changes as well as the manner in which the debate was unfolding. Whatever the truth of the matter, one thing was clear: For a technological standard that had become in essence a shared property of the Internet, there was not enough consensus to move it forward. There would not be enough adoption of this new standard for it to be a viable technology.

Instead of throwing more energy behind a process that wasn't working, a second group was formed, primarily consisting of Microsoft and Yahoo! members, that worked in parallel to come up with a more modest revision to the standard addressing some immediate needs that were not as contentious. This standard would be known as ES3.1 (Edition 3.1) and then later renamed to ES5. This was initially intended to be a halfway marker between what was appearing in the ES4 document and what was already implemented in ES3. This has been described by some as more of a bugfix than a major update to the standard.

These two groups attempted to coordinate their efforts such that changes made in ES3.1 would be carried forward to ES4. However, as a result of fundamental differences of opinion, it became clear that this was not going to happen and that there was too little common ground. Again, progress was at a standstill. Already, Adobe had adopted ES4 in the latest version of their engine for Flash and Flex development (ActionScript 3). Now it looked as though there was no future for ES4 as it was currently described.

Finally, it was decided that the two groups had to come together with more modest ambitions so that everyone could move forward. This new and completely separate project would be known as ECMAScript Harmony and would retain little of what was originally planned for ES4. Although a published draft became available in early 2009, it will probably be years before developers can rely on the features of ES5 in most browsers.

Stages of a JavaScript Developer

Despite the current popularity of JavaScript in the browser, it's actually very difficult to find developers who understand it well. This is true in any job market, whether it be the Bay Area, the deeply digital tech sector of Vancouver, or even the highly professional New York developer community. This is fundamentally because the interconnectedness between JavaScript and related technologies (CSS, HTML, and the browser) creates deep complexities that only an enthusiast can fully master. It's also because a thorough understanding of server technologies and transport formats like XML, JSON, and SOAP is often required. Rarely will you see a job posting for a "JavaScript" expert but instead for a multi-discipline developer experienced in JavaScript as well as many other technologies. As a result, some developers are choosing to become adept at one or more of the popular JavaScript *frameworks* such as *jQuery*, *Dojo*, *Prototype*, or *Mootools*. These are very practical ways to approach browser scripting, and I highly recommend learning one, but these frameworks are by nature minimalistic. They are not particularly forgiving if you lack an understanding of CSS, Object Orientation, or interacting with the document object model.

If you're new to the language, you're probably overwhelmed with the number of resources available for learning it. You may have read some other books or even tried your hand at some basic scripting. It's possible to rapidly accelerate your mastery of both JavaScript and browser scripting in general by familiarizing yourself with the fundamentals. If you already know another programming language and can become proficient with the four or so basic concepts in the language, you can say goodbye to months of gradual discovery and terrible code and jump right into the really fun stuff. John Resig of jQuery and Mozilla fame was one of the first to describe a common development path for new coders when learning

the language (`http://www.slideshare.net/jeresig/building-a-javascript-library`). It goes something like this:

- Object references are everywhere: Most useful operations involve passing references to very large objects like the DOM (Document Object Model) or an element on the page or a function. The DOM itself is a very large hierarchical collection of object and element references that can be manipulated as easily as setting a property.
- You can make your own objects and namespaces: Indeed, one of the first things developers realize is that JavaScript is OO (Object Oriented) programming. While they may not fully understand all the OO features available to them, they begin by making some basic APIs that follow very elementary OO principals.
- Object prototypes let you create OO classes: Once coders understand that they can create instances of objects and functions to build pseudo-classes, someone points out the *prototype* constructor to them and somewhere in the learners' brains a light goes off. They begin building elaborate class-based APIs for every imaginable purpose but begin hitting roadblocks related to scope and maintaining object references between pieces of their programs.
- Closures are God: As Resig pointed out in his now-famous talk, at this stage coders generally discover how closures can help solve some of the problems encountered in stage 3 when building complex interconnected APIs. They may not, however, fully understand the minefield that closures are. Memory leaks, difficult-to-follow scope chains, and spaghetti code are coexistent with a coder's first attempts at closures.

Real-World JavaScript

As was touched on already, the JavaScript language (most often referred to by its ECMA name, *ECMAScript*) crops up all over the place — not just in web pages. It also takes surprisingly different forms depending on where it's used. To provide a complete context for the landscape of ECMAScript use, here are some examples of these uses.

In the Browser

Browser-based development is certainly the original and predominant platform for JavaScript. JavaScript can be executed in the context of a web page or even in the form of a browser plug-in in the case of Firefox plugins. Web developers certainly have a lot to contend with. First and foremost, they've got to decide which browsers and platforms make up their audience. If they're developing sites for desktop browsers, at least three targets should be tested: Internet Explorer, Firefox, and Safari. They'll also want to test all of the most popular versions of these browsers (which usually doesn't mean the *latest* version). For most purposes, the core language of JavaScript differs little among the latest versions of these, and thankfully they function much the same way whether they're running on a Mac or PC. Where it gets a little complicated is if they want to include mobile platforms as well, cell phones and gaming consoles in particular. A lot of cell phones use the *Opera* browser platform, as does the *Nintendo Wii* browser. Blackberry phones use their own proprietary browser and JavaScript engine, and Apple's *iPhone* uses a trimmed down version of Safari.

One of the key considerations when writing JavaScript for mobile platforms is the abysmal performance offered by these devices, as illustrated by the graph in Figure 1-1. In these cases, it becomes even more important to use best practices for high-performance code. Many of these are described in Chapter 25.

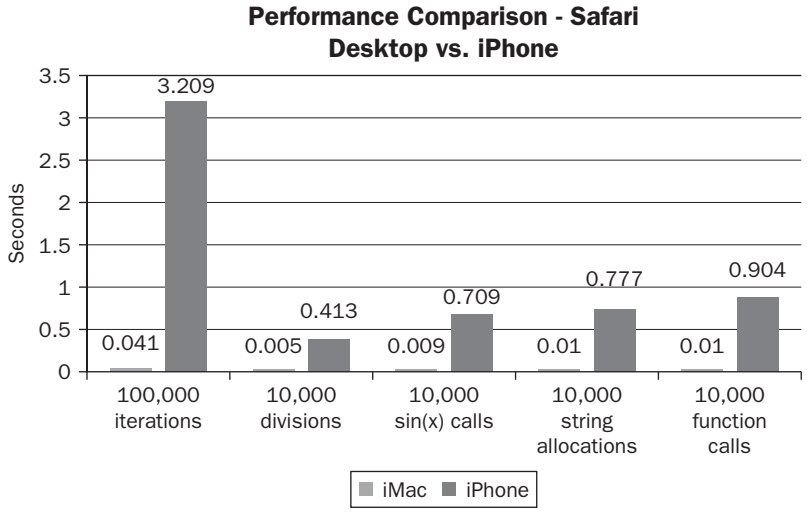

Figure 1-1

When used in a browser, JavaScript is considered an *interpreted language*. This sets it apart from other programming languages such as C++, a *compiled language*. When a browser downloads a page with JavaScript embedded, it receives the original source code of the script. It then passes the script to a program called an interpreter, which converts it to machine code on the fly. The browser does this every time it loads the page and does not attempt to cache or validate the program before it is executed. Errors are passed on to the user as they occur. The advantage for the developer is that it is a very lightweight way to write applications and the main debugging environment is the browser itself. The disadvantage is that all your source code is visible to anyone that wants to see it. Also, because it is interpreted on the fly, not compiled to machine code first, JavaScript is not suitable for writing CPU-intensive applications like a 3D game or Autocad program, mainly because it won't be fast enough.

Server-Side JavaScript

Although the development context is different, JavaScript has also been implemented many times over as a *server-side scripting language*, often to generate web pages. This was first done by Netscape as part of their *Enterprise Server 3.0* product in the form of a feature called *LiveWire*. That was in 1996. Today there are many server-side frameworks implementing JavaScript. Some of these use open source interpreters such as *Rhino* or *SpiderMonkey*. Microsoft uses their interpreter (called *JScript*) in both their browser and their development runtime *.NET*. Even the now-obsolete *ASP* framework from Microsoft had JScript as

an available language. Today, very few people choose JScript when writing applications in .NET, but it lives on in products from other vendors:

Name	Javascript Engine	More Information
AppJet	Rhino	http://www.appjet.com/
Aptana Jaxer	SpiderMonkey	http://www.aptana.com/jaxer/
ASP	JScript	From Microsoft (Now Obsolete)
ASP.NET	JScript.NET	http://msdn2.microsoft.com/en-us/library/ms974588.aspx
Cocoon Flowscript	Rhino	http://cocoon.apache.org/2.1/userdocs/flow/api.html
Helma Object Publisher	Rhino	http://www.helma.org/
jsext	SpiderMonkey	http://www.jsext.net/
JSP	Caucho Resin Servlet Runner V2	From Sun Microsystems (Now Obsolete)
JSSP	Rhino	http://jssp.de/
Junction	Rhino	http://code.google.com/p/trimpath/
mod_js	SpiderMonkey	http://modjs.org/
OpenMocha	Helma	http://openmocha.org/openmocha/
Phobos	Rhino	https://phobos.dev.java.net/
Rhino in Spring	Rhino	http://rhinoinspring.sourceforge.net/
Rhinola	Rhino	http://mod-gcj.sf.net/rhinola.html
Server Side JavaScript	Rhino	http://www.bluishcoder.co.nz/2006/05/server-side-javascript.html
10gen	10gen Proprietary	http://www.10gen.com/
Torino	Rhino	http://torino.sourceforge.net/
Whitebeam	SpiderMonkey	http://www.whitebeam.org/
wxJavaScript	SpiderMonkey	http://www.wxjavascript.net/

ActionScript and Flash

Introduced in *Macromedia Flash Player* 5, *ActionScript* was an improvement on a scripting feature introduced into Flash much earlier. The idea was simply to allow developers to apply custom movements and behaviors based on user input. It was a full implementation of ECMAScript V1 and allowed for both *procedural* and *object oriented* development styles. Around the time Adobe acquired Macromedia, ActionScript 2.0 was released, which implemented ECMAScript working draft V4. It was anticipated that the browser vendors would eventually follow-suit and implement this version also. Unfortunately, subsequent political disputes between Microsoft and the Mozilla Foundation severely reduced the likelihood that this version would ever be adopted universally, making Adobe one of the few vendors likely to ever implement this particular branch of the language. Today, ActionScript is implemented in both Flash and Flex and has a huge following of professional developers.

Adobe Integrated Runtime (AIR)

Adobe Integrated Runtime (AIR) is a relatively new offering from Adobe, but is already an important fixture in the programming landscape. It offers cross-platform *write-once, run anywhere* desktop development with a special focus on ease of integration with web services. Developers can write applications in Flex or in HTML with JavaScript that can be compiled to run on OSX, Windows, or Linux desktops. The HTML / JavaScript implementation is achieved by repackaging a custom version of Webkit (Safari) with some API extensions to add features like online/offline detection, permanent SQLite storage, and multimedia support.

In Other Adobe Products

Adobe has also implemented JavaScript as the language used to script and customize products such as Dreamweaver (for making plugins), Acrobat (for customizing interfaces), and InDesign.

Desktop Widgets

With the popularization of Apple Dashboard widgets, Konfabulator widgets from Yahoo, and Microsoft Gadgets for Vista, it's now clear that JavaScript is the language of choice for desktop and dashboard-type gadgets. A widget can be an egg timer, a news reader, or even a simple game. In each of these cases, widgets can be generally constructed using a combination of JavaScript, CSS, HTML, and/or XML. Depending on the platform, they may have some limited access to system resources (like the file system), but generally they run in the context of a very small webpage. Apple Dashboard widgets have the added capability of using Canvas (graphical) elements because they are rendered using Safari's browser engine WebKit.

Complementary Technologies

In the world of browser scripting in particular is a set of complementary technologies that developers must understand. In this book, I will refer to these a great deal and you will develop a thorough understanding of how they fit into the development stack and how developers can use them in their applications to build powerful interfaces.

Hypertext Markup Language (HTML)

The declarative document markup language that makes up a web page interacts extensively with JavaScript. Script allows us to make the page *dynamic* by writing new contents, and modifying existing contents. You can interact with HTML by treating it as a big string and working with all the words and symbols that make it up (as in the second example below), or by using the *DOM* (Document Object Model) to manipulate the page in a hierarchical object-based way. Using HTML, you can tell the browser to execute a block of script inline with the page using the following syntax:

```
<html>
    <head>
    <script type="text/javascript">
        // This JavaScript block will execute first
    </script>
    </head>
<body>
    <script type="text/javascript">
        // This block will execute second
    </script>

    <h1>Hello World</h1>

    <script type="text/javascript">
        // This block will execute last
    </script>
</body>
</html>
```

You can also use JavaScript to generate HTML by simply writing it to the page:

```
<html>
<body>
    <script>
        document.write("<h1>Hello world!</h1>");
    </script>
</body>
</html>
```

Cascading Style Sheets (CSS)

CSS describes the color, size, position, and shape of most things on a web page. CSS documents can statically describe the look and feel of a document, but these attributes can also be changed after the page has loaded. There is an in-depth object model available to script developers who wish to use it to dynamically modify these attributes on the fly. By manipulating the style of an element with script, you can animate its size or position, have it move in front of or behind other elements, or make it fade away to nothing.

In the following example, you change the color of the document by modifying the background color CSS attribute of the document object (it's ok if you don't understand this yet).

```
<html>
  <body>
  <script type="text/javascript">
    document.body.style.backgroundColor = 'green';
  </script>
  </body>
</html>
```

You can see this rendered in Internet Explorer in Figure 1-2.

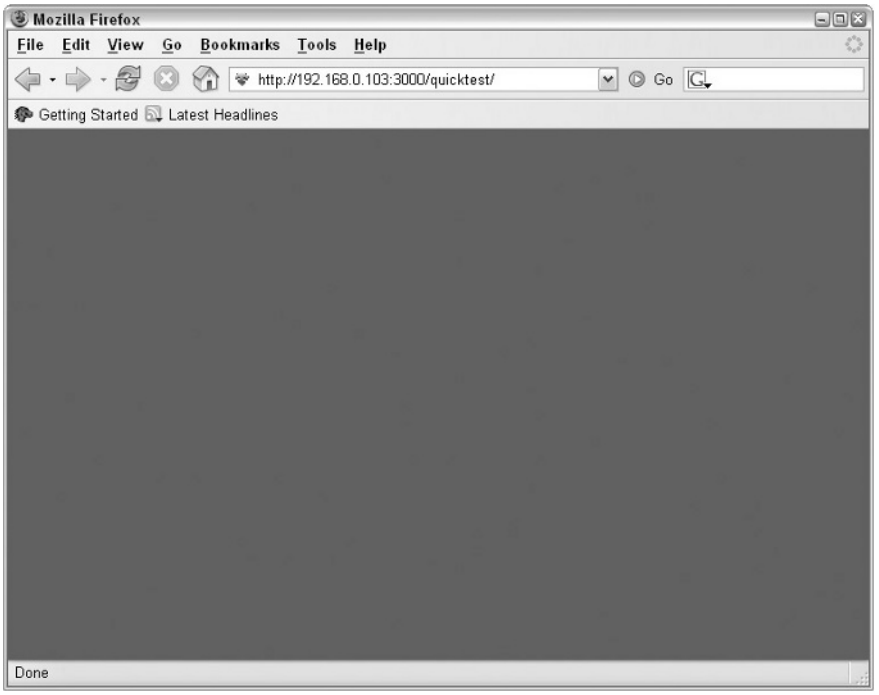

Figure 1-2

The Browser Object Model (BOM)

JavaScript in a browser is essentially a group of object models relating to specific areas of functionality within the browser. One of these is known as the BOM (Browser Object Model), which represents the browser itself. The browser object can be accessed by referencing the top-level object `window`. From here you can access things such as the `document` object, the `frames` collection, the browser history, the status bar, and so on. In large part, what you find in the BOM depends on what browser you are operating in. However, the main pieces can be seen in Figure 1-3.

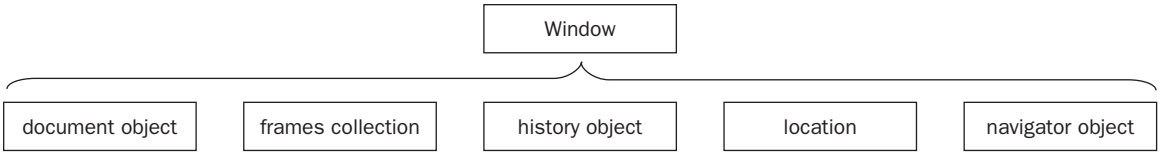

Figure 1-3

The Browser Object Model consists of the following sub-components:

- The `document` object: Represents the document object of the current page.
- The `frames` collection: Provides array of the frames within the current page.
- The `history` object: Contains the history list of the browser.
- The `location` object: Holds the current URL that the browser is displaying.
- The `navigator` object: Has information about the browser itself, like the version number, and browser engine.

The Document Object Model (DOM)

By far the most important object of all the available object models in a browser is the `document`. The document gives access to all the elements on a page as a hierarchical collection of nodes. It also contains some meta information about the page itself such as title and URL and gives access to some short-hand collections of common elements like forms, links, and `<a>` tags (anchors). The document object can be accessed from any part of a JavaScript application from `window.document`, or simply `document`.

The DOM is a very large object but some of the most common top-level properties can be found as follows. A more complete reference with full browser support information is in Appendix F.

Document Property	Description
`body`	Returns a reference to the `<body>` container of the current page, containing all the HTML on the page.
`cookie`	Gives read and write access the cookies accessible by this page.
`forms[ ]`	An array of all the forms on the page, including all the form fields within them.
`links[ ]`	An array of all the hyperlinks on the page.
`location`	Gets and sets the location, or current URL, of the window object.
`title`	The title of the document (defined in the `<title>` tag).

You'll look at this in more detail later, but for now it's enough to know that the document is a representation of the current page, is dynamic (can be modified via JavaScript calls), and is not exactly the same between browser engines. For example, Internet Explorer document object contains methods

and properties not available in WebKit's, and vice-versa. Over time, the object models have evolved considerably too. The first DOM (supported in Netscape 2 and Internet Explorer 3) supported only a small fraction of what is available today. In Appendix F you can find detailed browser support data with version information to assist you.

When to Use JavaScript

Sometimes it's useful to consider the "big picture" when looking at a new technology. Once you understand that, you can begin to anticipate the answers to other questions that might come up. One of those "big picture" questions for JavaScript is what *can* it do and what can it *not do*. After all, it is a scripting language and running inside a browser (usually) — we know there must be limits to its power.

Let's start with the types of things it *can do*:

- **Dynamically draw boxes, images, and text on the page:** Using Dynamic HTML and the DOM, you can arbitrarily style and animate these types of objects on a webpage.
- **Open and close windows:** You can spawn new browser windows and communicate with them to some degree. You can also create *simulated* windows using DHTML and even provide drag and drop support for them like real windows that would appear elsewhere on the desktop.
- **Animate on-screen contents:** You can create multiple, simultaneous, threaded animations using DHTML, the DOM, and JavaScript timers.
- **Modify the document:** You can create elements, text, and images, or you can delete or modify existing ones.
- **Communicate with the server:** Using Ajax and similar techniques, you can asynchronously send messages back and forth between the server and the client without forcing the page to re-load.
- **Talk to Java, Flash, Silverlight objects:** You can even communicate with other types of media embedded on the page to control the behavior of Flash and Silverlight movies or interface with Java Applets.
- **Snoop on the user; record what they do:** Yes, it's even possible (however nefarious) to record everything your website users are doing on a page, their mouse movements, keystrokes, and so on, and study it later. There are benign uses for this data, too (for example web analytics).
- **Read the mouse/keyboard:** You can keep detailed track of what the user is doing with the keyboard and mouse in order to create extremely rich and interactive web applications.
- **Save data offline for later:** You can put information in semi-permanent storage on the user's computer so that the next time they come to our page and want access to it, they can have it.
- **Create free-form graphic elements:** Using complementary technologies like Canvas elements, Scalable Vector Graphics, and Flash, you can put free-form elements on a page and even change them on the fly.
- **Create accessible web pages:** A common misconception is that it is not possible to have an accessible web page for people with disabilities and still use JavaScript. Most web users with disabilities are using browsers that do support JavaScript. Given a bit of care and attention, you can make sure your pages are easy for them to use.

Things you can't do with JavaScript in a browser:

- **Manipulate files on the file system:** A script cannot arbitrarily open and read files on a user's hard drive.
- **Talk directly to hardware:** You can't write a program that interacts directly with game controllers or other external hardware.
- **Read freely from memory:** You can't access anything in the computer's memory beyond its immediate stack of local variables and what's in the object model.
- **Perform general networking:** Without using a plugin, you can't open sockets or perform general networking tasks beyond what is possible via a simple HTTP request.
- **Interact with the desktop:** JavaScript cannot be used to open or close windows or programs on the users' desktops, unless they are browser windows.
- **Open windows that are too small:** As a result of unwholesome techniques employed by online advertisers in the past, there are tight restrictions on the sizes and positions of windows that can be opened by JavaScript calls.
- **Set the value of FileUpload fields:** Owing to security restrictions, you can't set the value of FileUpload form fields.
- **Provide access to rich media without the use of plugins:** Although I previously stated you could create free-form graphic elements on the fly, this is only partly true. It's true when users have the necessary plugins available (i.e.: Flash). Otherwise, you would not be able to play a sound or movie file or do rich graphics on the page.

The simple answer to when you should use JavaScript is that you should use it whenever you deem that nobody in your audience is harmed (through compatibility problems, accessibility, or performance) and indeed most people will benefit. While this is a decidedly cryptic response, the truth is that there is a lot you can do with JavaScript to make applications easier, faster, and more enjoyable for everybody. Provided you are using the least-harm philosophy in your development, aim high — your users will thank you!

Major JavaScript Engines

The feature inside a browser that interprets all the JavaScript on a page is called the *JavaScript Engine*. This is different from the feature inside a browser that renders HTML and CSS, which is known as the *Layout Engine*. There are nearly as many engines as there are browser vendors. This has been the source of a lot of confusion over the years because more often than not, it is less important to a developer which browser someone is using than it is what JavaScript or Layout engine they are using.

Generally speaking, JavaScript engines implement a single or sometimes multiple versions of the ECMAScript standard. With few exceptions, most modern engines are compliant up to edition 3 of this standard. However, all the major vendors (Microsoft, Mozilla, WebKit) have developed custom extensions to the language and to the object models which are in varying stages of support in competing engines despite the fact that they are not part of the "official" standard.

In some cases (for example, in the case of Mozilla's *Rhino* and *SpiderMonkey* engines), the code implementing JavaScript is modular and can be used by third parties outside of a browser. This has

resulted in a massive propagation of JavaScript-supported scripting tools using some common platform — so it has become more important than ever to keep track of which engines support which features of the language and how they compare to one another.

Following is a list of major JavaScript engines.

Engine	Vendor	Description
Rhino	Mozilla	An open source JavaScript implementation written in Java. Supports up to v1.7 of Mozilla's JavaScript. `http://www.mozilla.org/rhino/`
SpiderMonkey	Mozilla	The first-ever JavaScript engine, still in use today in Mozilla-based browsers (Netscape 6+ and Firefox 1+). Written in C++, it is embedded in many 3rd party applications, much like Rhino and JavaScriptCore. Most recent versions include the ground-breaking TraceMonkey features supporting on-the-fly byte-code compilation and optimization of code for improved performance.
JavaScriptCore	Webkit	Open source and originally derived from KDE's JavaScript engine. It's currently used in the Webkit browser engine which is implemented in Apple's Safari (superceded by SquirrelFish Extreme in 2008), Adobe AIR, iCab 4+, Konqueror, and Flock among others. Also used in the OSX operating system for some scripting features, and in Dreamweaver CS4 to provide in-IDE testing of JavaScript.
SquirrelFish	Webkit	An incremental rewrite of JavaScriptCore to be implemented in most new versions of Webkit-based browsers including Safari. `http://trac.webkit.org/wiki/SquirrelFish`
JScript	Microsoft	A component of Microsoft's Trident layout engine and used in all versions of Internet Explorer after 3.0. Also used as a component in Windows, ASP, and the .NET programming framework.
Tamarin	Adobe	A free (GPL, LGPL, and MPL) ECMAScript engine used in Flash v9.0 and up. Implements the Adobe language known as ActionScript, which is primarily an implementation of ECMAScript.
V8	Google	An open-source (BSD) ECMAScript engine used in Google's Chrome browser (which is based on WebKit). `http://code.google.com/p/v8/`
Elektra	Opera	The proprietary layout and JavaScript engine used by the Opera browser versions 4-6.
Presto	Opera	The proprietary layout and JavaScript engine used more recent versions of Opera (7.0+). This engine is also implemented on many of the mobile and platform devices that support Opera, such as Nintendo Wii, and Nintendo DS. Adobe's Dreamweaver (up to CS4) uses Presto.

ECMAScript Support by Engine

ECMAScript is the Ecma International specification that describes the JavaScript language. Strictly speaking, browsers implement ECMAScript, the standard, not JavaScript, a Sun trademark licensed by Mozilla and the name of Mozilla's engine.

There have been four revisions to the original ECMA 262 draft, all at different stages of adoption:

Edition	Published	Differences to the Previous Edition
1	June 1997	n/a
2	June 1998	Editorial changes to keep the specification fully aligned with ISO/IEC 16262 international standard.
3	December 1999	Added regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and other enhancements.
3.1 (Now 5)	Work in progress	AKA "Harmony." More modest syntactic changes than revision 4. Class-based inheritance.
4	Work in progress	Multiple new concepts and language features. Has been since superseded, oddly enough, by the newer v3.1.

ECMAScript Support corresponds to different Engine versions as follows:

ECMAScript Edition	Mozilla JavaScript Edition	Microsoft JScript Version
1st Edition	1.3 (Netscape 4.06-4.7x, October 1998)	3.0 (IE 4.0, Oct 1997)
2nd Edition	1.3 (Netscape 4.06-4.7x, October 1998)	3.0 (IE 4.0, Oct 1997)
3rd Edition	1.5 (Netscape 6)	5.5 (IE 5.5, July 2000)
3.1 Edition (Now 5.0 Edition)	Unknown	Possibly 5.9 (Predictive)
4.0 Edition	2.0 (In progress and in question)	Probably never, unless 3.1 becomes 4.0

General Equivalence

If you follow the changes introduced in JavaScript engines over time, you can compare browsers generally in terms of JavaScript object model equivalencies and support of the ECMA standard. If you are to infer anything from the chart that follows, it might be that years of cooperation by browser vendors has begun to break down in recent times. Rapid evolution of the Firefox browser in particular has made it hard for the others to keep pace. With the recent introduction of the ECMAScript 3.1 draft, you may see less rapid innovation in the future in lieu of cooperation, at least in terms of object models and APIs, if not in other aspects of the engine such as performance.

JavaScript Ver.	JScript Ver.	ECMA Ed.	IE Ver.	Netscape Ver.	Firefox Ver.	Opera Ver.	Safari Ver.	Chrome
1.0	1.0	Pre	3.0	2.0	n/a	n/a	?	n/a
1.1	2.0	Pre	n/a	3.0	n/a	n/a	?	n/a
1.2	3.0	Pre	4.0	4.0-4.05	n/a	n/a	?	n/a
1.3	3.0	1 and 2	4.0	4.06-4.7	n/a	n/a	?	n/a
1.3	4.0	1 and 2	n/a	n/a	n/a	n/a	?	n/a
1.4	5.0-5.1	1 and 2	5.0-5.01	Server only	n/a	n/a	?	n/a
1.5	5.5	3	5.5	6.0	1.0	6.0-9.0	?	1.0
1.5	5.6	3	6.0	6.0	1.0	6.0-9.0	?	1.0
1.5	5.7	3	7.0	6.0	1.0	6.0-9.0	?	1.0
1.5	5.8	3	8.0	6.0	1.0	6.0-9.0	?	1.0
1.6	n/a	3	n/a	7.0	1.5	n/a	?	n/a
1.7	n/a	3	n/a	8.0	2.0	n/a	3.0	n/a
1.8	n/a	3	n/a	n/a	3.0	n/a	n/a	n/a
1.9	n/a	3	n/a	n/a	3.1	n/a	n/a	n/a

Performance in JavaScript Engines

Comparing JavaScript engines is a dodgy business. The choice of operating system and exactly what kind of test is run can greatly influence results. Still, you can learn something from benchmarks, if only that browsers are getting faster. In Figure 1-4 are the results of the SunSpider benchmark tool on a number of recent browsers.

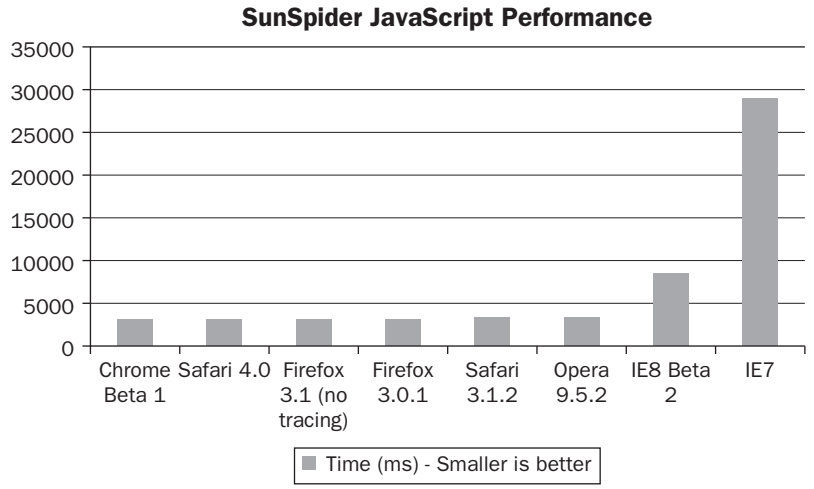

Figure 1-4

What we are seeing in the browser engines lately is new emphasis on the raw performance of the JavaScript interpreter — something not considered a major issue before. This is the natural outcome of an increasing reliance on JavaScript-rich web applications. They are achieving this in part by treating the interpreter as a compiler of sorts and running highly sophisticated analysis on the code to generate the most concise byte-code possible. The result is an interpreted JavaScript application with performance approaching that of native compiled code.

In Figure 1-5 you can see how Safari (WebKit) JavaScript performance has improved lately.

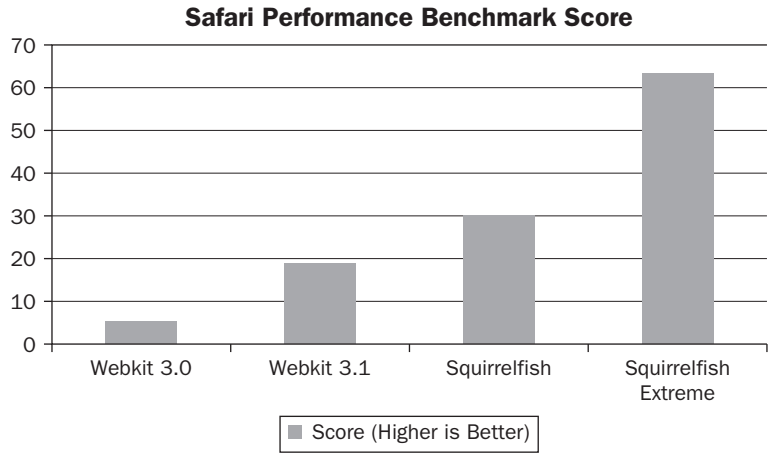

Figure 1-5

Basic Development Tools

All that you need to develop JavaScript applications is a text editor and a web browser. Here you'll find some useful recommendations for each, but if all you have is Windows Notepad and Internet Explorer, you can easily write and test the examples in this book.

Choosing a Text Editor

Some people prefer to work within the black box of a WYSIWYG (What You See Is What You Get) editor like Visual Studio when it is in design mode or Dreamweaver design mode. Microsoft FrontPage also provides this functionality. I strongly warn against getting comfy with this type of tool, because it generally does not accurately predict browser behavior and because you will need to spend most of your time looking at the actual code anyway. However, both Visual Studio and Dreamweaver are fine choices if you use only the text editing features. If you don't want to shell out for these programs (and your employer will not), here are some alternatives:

- **Aptana** (`http://www.aptana.com`): An Eclipse-based IDE with built-in Intellisense for help remembering those pesky method and property names, as well as a CSS helper for styling pages. There is a full free version, which is the one that most people use. For less than $100, you can upgrade to the pro version, which also has some support for debugging JavaScript applications right inside the IDE. *Mac*, *Windows*, *Linux*.
- **Microsoft Visual Web Developer Express Edition** (`http://www.microsoft.com/express/webdevelopment/`): A full-featured IDE based on Visual Studio and tailor made for web development. The especially useful thing about this one is you can configure it to debug your JavaScript code outside of a browser. If you can't afford Visual Studio but like those products, definitely consider this one. *Windows*.
- **Notepad++** (`http://notepad-plus.sourceforge.net`): Is an open source and free text editor intended for use as an IDE. Although fairly bare bones with no intellisense, it has excellent syntax highlighting and can even synchronize your project with a remote FTP or SSH server via an extensive plugin library. *Windows.*
- **Textmate** (`http://macromates.com/`): Called the "missing editor" for OSX, Textmate is the IDE of choice for developers on the Mac. Although at first glance this looks just like a text editor, as you dig in you will find a world of useful macros and snippets to assist you. This is not a free product but costs only about $50. *Mac.*

The Web Server

Although not required, it may be helpful down the road if you are developing with the context of a web server on your machine, if your pages are simple static HTML with some JavaScript (and no Ajax), this is not required. Simply point your browser to the page on your computer by using the *file:///* directive in

the address bar. If you are running Internet Explorer, you may get a security warning prompt, as in Figure 1-6. When running JavaScript off the file system in IE, you are running in a different security sandbox with tighter restrictions on active content. You can change your browser settings or just allow the content on that page by clicking the button and choosing Allow Blocked Content.

Ultimately, you're going to want to set up a web server on your computer to do proper testing of Ajax RPCs. On Windows you can set up the free Internet Information Services (IIS) server by first installing it from the Control Panel ⇨ Add/Remove Programs. You should be able to put HTML documents in `C:\inetpub\wwwroot\` and view them in your browser by surfing `http://localhost`.

On OSX is a built-in Apache web server that can be activated from the System Preferences application by clicking Sharing and selecting Web Sharing.

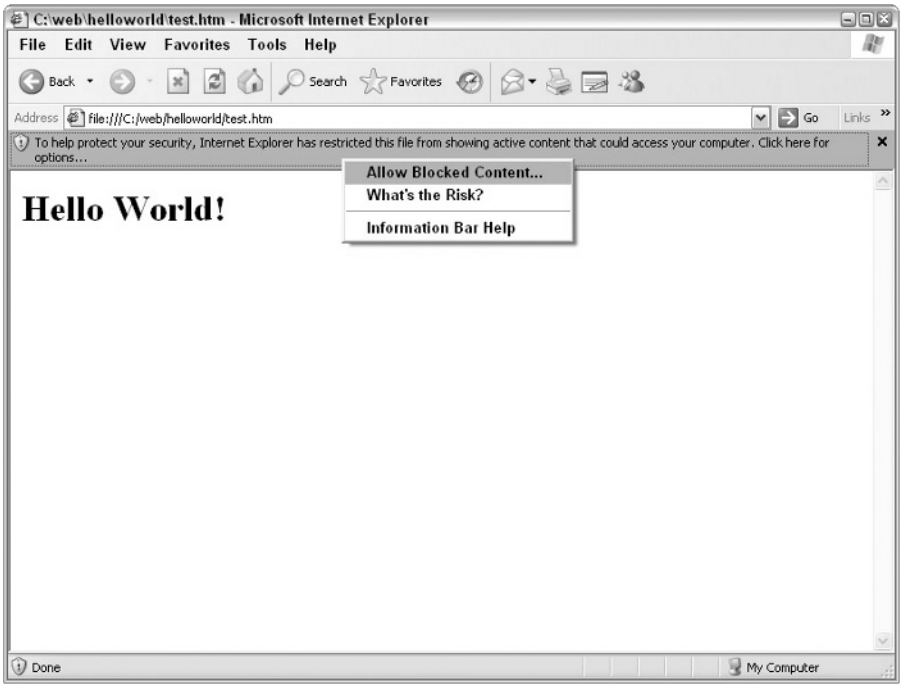

Figure 1-6

The Test Browser

Once you've got your IDE and your web server set up (if indeed you want to have a web server), make sure you've got a good cross-section of browsers to test with. The latest numbers (November 2008) report that Internet Explorer, Firefox, and Safari should be on your list for testing. These provide good coverage of the marketplace, and if your code runs in these, they will most likely run in newer versions of Opera, Netscape, and Google Chrome.

On Windows, Internet Explorer comes pre-installed. You can choose to upgrade to the latest version or leave it the way it came. You should then also download Firefox from `http://www.getfirefox.com` and Safari from `http://www.apple.com/safari/`. Google Chrome can be downloaded at `http://chrome.google.com` and Opera from `http://www.opera.com`.

On Mac, you'll want to download Firefox from the same location, and, of course, Safari comes pre-installed. For testing Internet Explorer, we suggest you run copies inside a Windows VMWare or Parallels image right on your desktop.

In Chapter 21, I'll talk more about tools that can assist you in debugging your applications inside a browser. For now, just make sure you can load a test page on your computer using whatever browser you have handy at least by using the file:// technique mentioned earlier.

Your First JavaScript Application

This chapter provides a lot of background on the history and role that JavaScript plays in development, but no introduction on a programming language would be complete without one bare-bones example. Remember that all the examples in this book can be found online at `http://wroxjavascript.com`.

There are several ways to augment a web page with JavaScript. One is to use the HTML tag `<script>` to indicate a portion of the page for script. This is known as a *script block*. When a browser spots a script block in a page, it does not draw its contents to the page. Instead, it "parses" its contents as a script block in the *order that it appears*. Generally speaking, if there are two script blocks on a page, the top one will execute first. You are allowed to put script blocks in the `<head>` area of the page and also in the `<body>` area. Blocks in the header execute before ones in the body.

I'll talk more about `<script>` tags in Chapter 3 because there are a few more things you should know about them. For now, take a look at the HTML page that follows with some in-line JavaScript code.

```
<html>
  <body>
  <h1>Hello World!</h1>
  <script type="text/javascript">
    var today = new Date();
    document.write("<p>Today is: " + today.toString() + "</p>");
  </script>
  </body>
</html>
```

If you were to write this to a text file, save it to your hard drive, and load it in your browser, Figure 1-7 is what you would likely see:

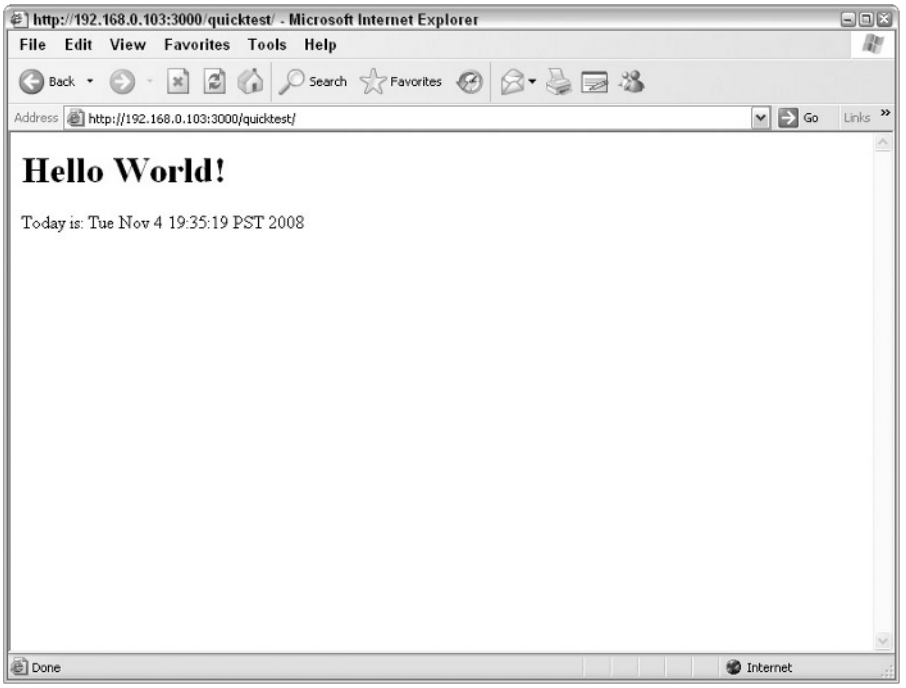

Figure 1-7

Let's take a look at the contents quickly to see what is happening.

Breaking it Down

I positioned the script block below the `<h1>` heading tag, so the browser executed it *after* it had rendered what came before it. The script block itself was ignored by the HTML layout engine, and its contents were passed on to the JavaScript engine within the browser. Looking at the first line of code:

```
var today = new Date();
```

What you see here is known as a *statement*. Every line of code in JavaScript is called a statement, in fact. To be precise, you should know that you can put many statements on a line of code, as long as they are separated by a semicolon. For legibility I have put each statement here on a separate line and have also used generous indentation — something done purely for cosmetic reasons.

Getting back to this particular line of code, I've used the `var` statement to declare a variable. I'll go into this in more detail in Chapter 2. For now it's enough to know that I declared a variable and assigned a value to it — a new instance of the global `Date` object. It just so happens that in JavaScript, when you create a date and do not say specifically *which date*, it automatically becomes today's date and time. This is what I've done here.

Moving on to the next statement in our block:

```
document.write("<p>Today is: " + today.toString() + "</p>");
```

Earlier in this chapter I spoke a bit about the DOM (Document Object Model). The `document` referred to here is, in effect, the page that you see. There happens to be a method on this object called `write()`, which allows us to append some text to it. This text may or may not contain HTML tags. In my case, I create a paragraph tag in the text I output, and I also output the value of my variable.

By using the plus + operator, you can easily concatenate multiple strings together. Here, the three strings are `"<p>Today is:"` whatever is output by `today.toString()`, and `"</p>"`. The JavaScript engine evaluates this operation before passing it on to `document.write` and outputting it to the page.

That's all there is to this particular program. After the engine encounters the closing `</script>` tag, it passes any result on to the layout engine and carries on rendering the remainder of the page.

Summary

By now, you know a lot about how JavaScript got to the place it is today and what you can achieve with it. You have learned that:

- JavaScript evolved gradually from a fairly primitive Netscape scripting extension to a sophisticated tool supported across the industry. The language continues to progress and change, and the shape it will take in the future is not entirely certain.
- ECMAScript, the standard that JavaScript is based on, has been implemented outside of the browser in many different technologies, including Microsoft's .NET, Adobe Flex and Flash, and even on the desktop.
- Along with HTML, CSS, and others, JavaScript is just one piece in a sophisticated stack of technologies that work together. The differences in these technologies among browsers make JavaScript development challenging at times.
- No compiler is required to develop applications. Scripts are developed in a text editor and tested directly in the browser, making development more accessible.
- You were introduced to a basic stack of tools required to do development, including an IDE or text editor, a web server (needed for Ajax development in particular), and a browser test environment.
- Script blocks containing JavaScript code are executed generally in the order they appear in a web page.
- You were exposed to a basic "Hello World" type application that made use of script blocks, variables, and the document object.

In Chapter 2, I'll dig into how the JavaScript language fits into the browser context. I'll get more into the concept of the Document Object Model, explain the `<script>` tag, and talk about how and *when* JavaScript gets executed in a web page.