



JavaScript: The Definitive Guide, 4th Edition

By David Flanagan

E-Book Ripped by lilmeanman

Enjoy! <http://www.magic-html.com> -- Free Apps, E-Books, ETC

Dedication

This book is dedicated to all who teach peace and resist violence.

Preface

There have been many changes in the world of web programming with JavaScript™ since the third edition of this book was published, including:

- Second and third editions of the ECMA-262 standard have been published, updating the core JavaScript language. Conformant versions of Netscape's JavaScript interpreter and Microsoft's JScript interpreter have been released.
- The source code for Netscape's JavaScript interpreters (one written in C and one written in Java™) has been released as open source and is available to anyone who wants to embed a scripting language in his application.
- The World Wide Web Consortium (W3C) has published two versions (or levels) of a Document Object Model (DOM) standard. Recent browsers support this standard (to varying degrees) and allow client-side JavaScript to interact with document content to produce sophisticated Dynamic HTML (DHTML) effects. Support for other W3C standards, such as HTML 4, CSS1, and CSS2, has also become widespread.
- The Mozilla organization, using source code originally contributed by Netscape, has produced a good fifth-generation browser. At the time of this writing, the Mozilla browser is not yet at the 1.0 release level, but the browser is mature enough that Netscape has based its 6.0 and 6.1 browsers upon the Mozilla code base.
- Microsoft's Internet Explorer has become the overwhelmingly dominant browser on desktop systems. However, the Netscape/Mozilla browser remains relevant to web developers, especially because of its superior support for web standards. In addition, minor browsers such as Opera (<http://www.opera.com>) and Konqueror (<http://www.konqueror.org>) should be seen as equally relevant.
- Web browsers (and JavaScript interpreters) are no longer confined to the desktop but have migrated to PDAs and even cell phones.

In summary, the core JavaScript language has matured. It has been standardized and is used in a wider variety of environments than it was previously. The collapse of Netscape's market share has allowed the universe of desktop web browsers to expand, and JavaScript-enabled web browsers have also become available on non-desktop platforms. There has been a distinct, if not complete, move toward web standards. The (partial) implementation of the DOM standard in recent browsers gives web developers a long-awaited vendor-independent API to which they can code.

What's New in the Fourth Edition

This edition of *JavaScript: The Definitive Guide* has been thoroughly updated in light of the changes I just described. Major new features include complete coverage of JavaScript 1.5 and the third edition of the ECMA-262 standard on which it is based, and complete coverage of the Level 2 DOM standard.

Throughout the book, the focus has shifted from documenting particular JavaScript and browser implementations (JavaScript 1.2, Netscape 4, Internet Explorer 5, etc.) to documenting the standards upon which those implementations are (or ought to be) based. Because of the proliferation of implementations, it is no longer practical for any one book to attempt to document -- or for any one developer to attempt to understand -- every feature, proprietary extension, quirk, and bug of every implementation. Focusing on the specifications instead of the implementations makes this book easier to use and, if you take the same approach, will make your JavaScript code more portable and maintainable. You'll particularly notice the increased emphasis on standards in the new material on core JavaScript and the DOM.

Another major change in this edition is that the reference section has been split into three distinct parts. First, the core JavaScript material has been separated from the client-side JavaScript material ([Part IV](#)) and placed in a section of its own ([Part III](#)). This division is for the convenience of JavaScript programmers who are working with the language in an environment other than a web browser and who are not interested in client-side JavaScript.

Second, the new material documenting the W3C DOM has been placed in a section of its own ([Part V](#)), separate from the existing client-side JavaScript material. The DOM standard defines an API that is quite distinct from the "legacy" API of traditional client-side JavaScript. Depending on the browser platforms they are targeting, developers typically use one API or the other and usually do not need to switch back and forth. Keeping these two APIs distinct also preserves the organization of the existing client-side reference material, which is convenient for readers of the third edition who upgrade to this edition.

In order to accommodate all the new material without making the book much, much larger, I've gotten rid of reference pages for the trivial properties of objects. These properties are already described once on the reference page for the object, and putting another description in a reference page of its own was redundant and wasteful. Properties that require substantial description, as well as all methods, still have reference pages of their own. Furthermore, the design wizards at O'Reilly have created a new interior design for the book that remains easy and pleasant to read but takes up less space.

Conventions Used in This Book

I use the following formatting conventions in this book:

Bold

Is occasionally used to refer to particular keys on a computer keyboard or to portions of a user interface, such as the **Back** button or the **Options** menu.

Italic

Is used for emphasis and to signify the first use of a term. *Italic* is also used for email addresses, web sites, FTP sites, file and directory names, and newsgroups. Finally, *italic* is used in this book for the names of Java classes, to help keep Java class names distinct from JavaScript names.

Constant width

Is used in all JavaScript code and HTML text listings, and generally for anything that you would type literally when programming.

Constant width italic

Is used for the names of function arguments, and generally as a placeholder to indicate an item that should be replaced with an actual value in your program.

Finding the Examples Online

The examples printed in this book are available for download from the book's web site. Follow the Examples link from the book's catalog page:

<http://www.oreilly.com/catalog/jscrip4/>

Acknowledgments

Brendan Eich of the Mozilla organization is the originator and chief innovator of JavaScript. I, and many JavaScript developers, owe Brendan a tremendous debt of gratitude for developing JavaScript and for taking the time out of his crazy schedule to answer our questions and even solicit our input. Besides patiently answering my many questions, Brendan also read and provided very helpful comments on the first and third editions of this book.

This book has been blessed with top-notch technical reviewers, whose comments have gone a long way toward making it a stronger, more accurate book. Waldemar Horwat at

Netscape reviewed the new material on JavaScript 1.5 in this fourth edition. The new material on the W3C DOM was reviewed by Philippe Le Hegaret of the W3C; by Peter-Paul Koch, Head of Client-Side Programming at the Dutch Internet consultancy and creation company Netling Framfab (<http://www.netlingframfab.nl>); by Dylan Schiemann of SitePen (<http://www.sitepen.com>); and by independent web developer Jeff Yates. Two of these reviewers maintain useful web sites about web design with the DOM. Peter-Paul's site is at <http://www.xs4all.nl/~ppk/js/>. Jeff's site is <http://www.pbwizard.com>. Although he was not a reviewer, Joseph Kesselman of IBM Research was very helpful in answering my questions about the W3C DOM.

The third edition of the book was reviewed by Brendan Eich, Waldemar Horwat, and Vidur Apparao at Netscape; Herman Venter at Microsoft; and two independent JavaScript developers, Jay Hodges and Angelo Sirigos. Dan Shafer of CNET's Builder.Com did some preliminary work on the third edition. Although his material was not used in this edition, his ideas and general outline were quite helpful. Norris Boyd and Scott Furman at Netscape also provided useful information for this edition, and Vidur Apparao of Netscape and Scott Issacs of Microsoft each took the time to talk to me about the forthcoming Document Object Model standard. Finally, Dr. Tankred Hirschmann provided challenging insights into the intricacies of JavaScript 1.2.

The second edition benefited greatly from the help and comments of Nick Thompson and Richard Yaker of Netscape; Dr. Shon Katzenberger, Larry Sullivan, and Dave C. Mitchell at Microsoft; and Lynn Rollins of R&B Communications. The first edition was reviewed by Neil Berkman of Bay Networks, and by Andrew Schulman and Terry Allen of O'Reilly & Associates.

This book also gains strength from the diversity of editors it has had. Paula Ferguson is the editor of this edition and of the third edition. She's given the book a thorough and much-needed going over, making it easier to read and easier to understand. Frank Willison edited the second edition, and Andrew Schulman edited the first.

Finally, my thanks, as always and for so many reasons, to Christie.

—David Flanagan, September 2001

Chapter 1. Introduction to JavaScript

JavaScript is a lightweight, interpreted programming language with object-oriented capabilities. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers and embellished for web programming with the addition of objects that represent the web browser window and its contents. This client-side version of JavaScript allows executable content to be included in web pages -- it means that a web page need no longer be static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

Syntactically, the core JavaScript language resembles C, C++, and Java, with programming constructs such as the `if` statement, the `while` loop, and the `&&` operator. The similarity ends with this syntactic resemblance, however. JavaScript is an untyped language, which means that variables do not need to have a type specified. Objects in JavaScript are more like Perl's associative arrays than they are like structures in C or objects in C++ or Java. The object-oriented inheritance mechanism of JavaScript is like those of the little-known languages Self and NewtonScript; it is quite different from inheritance in C++ and Java. Like Perl, JavaScript is an interpreted language, and it draws inspiration from Perl in a number of places, such as its regular expression and array-handling features.

This chapter provides a quick overview of JavaScript; it explains what JavaScript can and cannot do and exposes some myths about the language. It distinguishes the core JavaScript language from embedded and extended versions of the language, such as the client-side JavaScript that is embedded in web browsers and the server-side JavaScript that is embedded in Netscape's web servers. (This book documents core and client-side JavaScript.) This chapter also demonstrates real-world web programming with some client-side JavaScript examples.

1.1 JavaScript Myths

JavaScript is the subject of a fair bit of misinformation and confusion. Before proceeding any further with our exploration of JavaScript, it is important that we debunk some common and persistent myths about the language.

1.1.1 JavaScript Is Not Java

One of the most common misconceptions about JavaScript is that it is a simplified version of Java, the programming language from Sun Microsystems. Other than an incomplete syntactic resemblance and the fact that both Java and JavaScript can provide executable content in web browsers, the two languages are entirely unrelated. The similarity of names is purely a marketing ploy (the language was originally called LiveScript; its name was changed to JavaScript at the last minute).

JavaScript and Java do, however, make a good team. The two languages have different sets of capabilities. JavaScript can control browser behavior and content but cannot draw graphics or perform networking. Java has no control over the browser as a whole but can do graphics, networking, and multithreading. Client-side JavaScript can interact with and control Java applets embedded in a web page, and, in this sense, JavaScript really can script Java (see [Chapter 22](#) for details).

1.1.2 JavaScript Is Not Simple

JavaScript is touted as a scripting language instead of a programming language, the implication being that scripting languages are simpler, that they are programming languages for non-programmers. Indeed, JavaScript appears at first glance to be a fairly simple language, perhaps of the same complexity as BASIC. JavaScript does have a number of features designed to make it more forgiving and easier to use for new and unsophisticated programmers. Non-programmers can use JavaScript for limited, cookbook-style programming tasks.

Beneath its thin veneer of simplicity, however, JavaScript is a full-featured programming language, as complex as any and more complex than some. Programmers who attempt to use JavaScript for nontrivial tasks often find the process frustrating if they do not have a solid understanding of the language. This book documents JavaScript comprehensively, so you can develop a sophisticated understanding of the language.

1.2 Versions of JavaScript

JavaScript has evolved over the years, and Netscape has released several versions of the language. Microsoft has released similar versions of the JavaScript language under the name "JScript." And ECMA (<http://www.ecma.ch>) has published three versions of the ECMA-262 standard that standardize the JavaScript language under the awkward name "ECMAScript."

[Table 1-1](#) lists these various versions and explains their key features and how they are related to one another. In this book, I often use the name "JavaScript" to refer to any implementation of the language, including Microsoft's JScript. When I'm specifically referring to ECMAScript, I often use the terms "ECMA-262" or "ECMA."

Version	Description
JavaScript 1.0	The original version of the language. It was buggy and is now essentially obsolete. Implemented by Netscape 2.
JavaScript 1.1	Introduced a true Array object; most serious bugs resolved. Implemented by Netscape 3.

Table 1-1. Versions of JavaScript

Version	Description
JavaScript 1.2	Introduced the <code>switch</code> statement, regular expressions, and a number of other features. Almost compliant with ECMA v1, but has some incompatibilities. Implemented by Netscape 4.
JavaScript 1.3	Fixed incompatibilities of JavaScript 1.2. Compliant with ECMA v1. Implemented by Netscape 4.5.
JavaScript 1.4	Implemented only in Netscape server products.
JavaScript 1.5	Introduced exception handling. Compliant with ECMA v3. Implemented by Mozilla and Netscape 6.
JScript 1.0	Roughly equivalent to JavaScript 1.0. Implemented by early releases of IE 3.
JScript 2.0	Roughly equivalent to JavaScript 1.1. Implemented by later releases of IE 3.
JScript 3.0	Roughly equivalent to JavaScript 1.3. Compliant with ECMA v1. Implemented by IE 4.
JScript 4.0	Not implemented by any web browser.
JScript 5.0	Supported exception handling. Partially compliant with ECMA v3. Implemented by IE 5.
JScript 5.5	Roughly equivalent to JavaScript 1.5. Fully compliant with ECMA v3. Implemented by IE 5.5 and IE 6. (IE 6 actually implements JScript 5.6, but 5.6 is not different from 5.5 in any way that is relevant to client-side JavaScript programmers.)
ECMA v1	The first standard version of the language. Standardized the basic features of JavaScript 1.1 and added a few new features. Did not standardize the <code>switch</code> statement or regular expression support. Conformant implementations are JavaScript 1.3 and JScript 3.0.
ECMA v2	A maintenance release of the standard that included clarifications but defined no new features.
ECMA v3	Standardized the <code>switch</code> statement, regular expressions, and exception handling. Conformant implementations are JavaScript 1.5 and JScript 5.5.

1.3 Client-Side JavaScript

When a JavaScript interpreter is embedded in a web browser, the result is client-side JavaScript. This is by far the most common variant of JavaScript; when most people refer

to JavaScript, they usually mean client-side JavaScript. This book documents client-side JavaScript, along with the core JavaScript language that client-side JavaScript incorporates.

We'll discuss client-side JavaScript and its capabilities in much more detail later in this chapter. In brief, though, client-side JavaScript combines the scripting ability of a JavaScript interpreter with the document object model (DOM) defined by a web browser. These two distinct technologies combine in a synergistic way, so the result is greater than the sum of its parts: client-side JavaScript enables executable content to be distributed over the Web and is at the heart of a new generation of Dynamic HTML (DHTML) documents.

Just as the ECMA-262 specification defined a standard version of the core JavaScript language, the World Wide Web Consortium (W3C) has published a DOM specification (or recommendation) that standardizes the features a browser must support in its DOM. We'll learn much more about this standard in [Chapter 17](#), [Chapter 18](#), and [Chapter 19](#). Although the W3C DOM standard is not yet as well supported as it could be, it is supported well enough that web developers can start writing JavaScript code that relies on it.

[Table 1-2](#) shows the core language version and DOM capabilities supported by various browser versions from Netscape and Microsoft. Note that the versions of Internet Explorer listed in the table refer to the Windows version of that browser. The capabilities of Macintosh versions of IE often vary (sometimes significantly) from the same-numbered versions for Windows. Also, bear in mind that IE allows the JScript interpreter to be upgraded independently of the browser itself, so it is possible to encounter an installation of IE that supports a version of the language greater than that shown here.

Table 1-2. Client-side JavaScript features by browser		
Browser	Language	DOM capabilities
Netscape 2	JavaScript 1.0	Form manipulation
Netscape 3	JavaScript 1.1	Image rollovers
Netscape 4	JavaScript 1.2	DHTML with Layers
Netscape 4.5	JavaScript 1.3	DHTML with Layers
Netscape 6 / Mozilla	JavaScript 1.5	Substantial support for W3C DOM standard; support for Layers discontinued
IE 3	JScript	Form manipulation

Table 1-2. Client-side JavaScript features by browser

Browser	Language	DOM capabilities
	1.0/2.0	
IE 4	JScript 3.0	Image rollovers; DHTML with <code>document.all[]</code>
IE 5	JScript 5.0	DHTML with <code>document.all[]</code>
IE 5.5	JScript 5.5	Partial support for W3C DOM standard
IE 6	JScript 5.5	Partial support for W3C DOM standard; lacks support for W3C DOM event model

The differences and incompatibilities between Netscape's and Microsoft's client-side versions of JavaScript are much greater than the differences between their respective implementations of the core language. However, both browsers do agree upon a large subset of client-side JavaScript features. For lack of better names, versions of client-side JavaScript are sometimes referred to by the version of the core language on which they are based. Thus, in client-side contexts the term "JavaScript 1.2" refers to the version of client-side JavaScript supported by Netscape 4 and Internet Explorer 4. When I use core-language version numbers to refer to client-side versions of JavaScript, I am referring to the compatible subset of features supported by both Netscape and Internet Explorer. When I discuss client-side features specific to one browser or the other, I refer to the browser by name and version number.

Note that Netscape and Internet Explorer are not the only browsers that support client-side JavaScript. For example, Opera (<http://www.opera.com>) supports client-side JavaScript as well. However, since Netscape and Internet Explorer have the vast majority of market share, they are the only browsers discussed explicitly in this book. Client-side JavaScript implementations in other browsers should conform fairly closely to the implementations in these two browsers.

Similarly, JavaScript is not the only programming language that can be embedded within a web browser. For example, Internet Explorer supports a language known as VBScript, a variant of Microsoft's Visual Basic language that provides many of the same features as JavaScript but can be used only with Microsoft browsers. Also, the HTML 4.0 specification uses the Tcl programming language as an example of an embedded scripting language in its discussion of the HTML `<script>` tag. While there are no mainstream browsers that support Tcl for this purpose, there is no reason that a browser could not easily support this language.

Previous editions of this book have covered Netscape browsers more thoroughly than Microsoft browsers. The reason for this bias was that Netscape was the inventor of JavaScript and (for a time, at least) held the dominant position in the web-browser

market. This bias toward Netscape has declined in each subsequent edition of the book, and the current edition is heavily focused on standards, such as ECMAScript and the W3C DOM, rather than on particular browsers. Nevertheless, readers may find that some of the original bias toward Netscape comes through in the material that remains from older editions.

1.4 JavaScript in Other Contexts

JavaScript is a general-purpose programming language; its use is not restricted to web browsers. JavaScript was designed to be embedded within, and provide scripting capabilities for, any application. From the earliest days, in fact, Netscape's web servers included a JavaScript interpreter, so that server-side scripts could be written in JavaScript. Similarly, Microsoft uses its JScript interpreter in its IIS web server and in its Windows Scripting Host product, in addition to using it in Internet Explorer.

Both Netscape and Microsoft have made their JavaScript interpreters available to companies and programmers who want to embed them in their applications. Netscape's interpreter was released as open source and is now available through the Mozilla organization (see <http://www.mozilla.org/js/>). Mozilla actually provides two different versions of the JavaScript 1.5 interpreter. One is written in C and is called "SpiderMonkey." The other is written in Java and, in a flattering reference to this book, is called "Rhino."

We can expect to see more and more applications that use JavaScript as an embedded scripting language.^[1] If you are writing scripts for such an application, you'll find the first half of this book, documenting the core language, to be useful. The web-browser specific chapters, however, will probably not be applicable to your scripts.

^[1] ActionScript, the scripting language available in Macromedia's Flash 5, is modeled after the ECMAScript standard, but it is not actually JavaScript.

1.5 Client-Side JavaScript: Executable Content in Web Pages

When a web browser is augmented with a JavaScript interpreter, it allows executable content to be distributed over the Internet in the form of JavaScript scripts. [Example 1-1](#) shows a simple JavaScript program, or script, embedded in a web page.

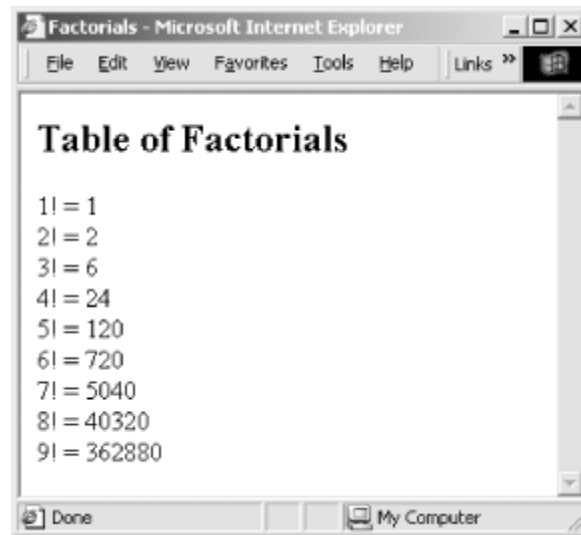
Example 1-1. A simple JavaScript program

```
<html>
<body>
<head><title>Factorials</title></head>
<script language="JavaScript">
document.write("<h2>Table of Factorials</h2>");
for(i = 1, fact = 1; i < 10; i++, fact *= i) {
    document.write(i + "! = " + fact);
    document.write("<br>");
}
```

```
}  
</script>  
</body>  
</html>
```

When loaded into a JavaScript-enabled browser, this script produces the output shown in [Figure 1-1](#).

Figure 1-1. A web page generated with JavaScript



As you can see in this example, the `<script>` and `</script>` tags are used to embed JavaScript code within an HTML file. We'll learn more about the `<script>` tag in [Chapter 12](#). The main feature of JavaScript demonstrated by this example is the use of the `document.write()` method.^[2] This method is used to dynamically output HTML text that is parsed and displayed by the web browser; we'll encounter it many more times in this book.

^[2] "Method" is the object-oriented term for function or procedure; you'll see it used throughout this book.

Besides allowing control over the content of web pages, JavaScript allows control over the browser and over the content of the HTML forms that appear in the browser. We'll learn about these capabilities of JavaScript in more detail later in this chapter and in much more detail later in this book.

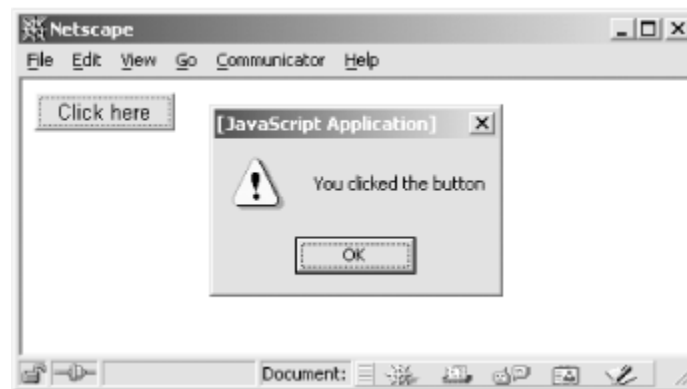
JavaScript can control not only the content of HTML documents, but also the behavior of those documents. That is, a JavaScript program might respond in some way when you enter a value in an input field or click on an image in a document. JavaScript does this by defining *event handlers* for the document -- pieces of JavaScript code that are executed when a particular event occurs, such as when the user clicks on a button. [Example 1-2](#) shows the definition of a simple HTML form that includes an event handler that is executed in response to a button click.

Example 1-2. An HTML form with a JavaScript event handler defined

```
<form>
<input type="button"
      value="Click here"
      onclick="alert('You clicked the button');">
</form>
```

[Figure 1-2](#) illustrates the result of clicking the button.

Figure 1-2. The JavaScript response to an event



The `onclick` attribute shown in [Example 1-2](#) was originally a Netscape extension added to HTML specifically for client-side JavaScript. Now, however, this and other event handler attributes have been standardized in HTML Version 4.0. All JavaScript event handlers are defined with HTML attributes like this one. The value of the `onclick` attribute is a string of JavaScript code to be executed when the user clicks the button. In this case, the `onclick` event handler calls the `alert()` function. As you can see in [Figure 1-2](#), `alert()` pops up a dialog box to display the specified message.

[Example 1-1](#) and [Example 1-2](#) highlight only the simplest features of client-side JavaScript. The real power of JavaScript on the client side is that scripts have access to a hierarchy of objects that are based on the content of the web page. For example, client-side JavaScript programs can access and manipulate each of the images that appear in a document and can communicate and interact with Java applets and other objects embedded within an HTML document. Once you have mastered the core JavaScript language, the key to using JavaScript effectively in web pages is learning to use the features of the DOM exposed by the browser.

1.6 Client-Side JavaScript Features

Another possible use of JavaScript is for writing programs to perform arbitrary computations. You can write simple scripts, for example, that compute Fibonacci numbers, or search for primes. In the context of the Web and web browsers, however, a more interesting application of the language might be a program that computed the sales

tax on an online order, based on information supplied by the user in an HTML form. As mentioned earlier, the real power of JavaScript lies in the browser and document-based objects that the language supports. To give you an idea of JavaScript's potential, the following sections list and explain the important capabilities of client-side JavaScript and the objects it supports.

1.6.1 Control Document Appearance and Content

The JavaScript Document object, through its `write()` method, which we have already seen, allows you to write arbitrary HTML into a document as the document is being parsed by the browser. For example, you can include the current date and time in a document or display different content on different platforms.

You can also use the Document object to generate documents entirely from scratch. Properties of the Document object allow you to specify colors for the document background, the text, and the hypertext links within it. This amounts to the ability to generate dynamic and conditional HTML documents, a technique that works particularly well in multiframe documents. Indeed, in some cases dynamic generation of frame content allows a JavaScript program to replace a traditional server-side script entirely.

Internet Explorer 4 and Netscape 4 support proprietary techniques for producing Dynamic HTML effects that allow document content to be dynamically generated, moved, and altered. IE 4 also supports a complete DOM that gives JavaScript access to every single HTML element within a document. And IE 5.5 and Netscape 6 support the W3C DOM standard (or at least key portions of it), which defines a standard, portable way to access all of the elements and text within an HTML document and to position them and modify their appearance by manipulating their Cascading Style Sheets (CSS) style attributes. In these browsers, client-side JavaScript has complete power over document content, which opens an unlimited world of scripting possibilities.

1.6.2 Control the Browser

Several JavaScript objects allow control over the behavior of the browser. The Window object supports methods to pop up dialog boxes to display simple messages to the user and get simple input from the user. This object also defines a method to create and open (and close) entirely new browser windows, which can have any specified size and any combination of user controls. This allows you, for example, to open up multiple windows to give the user multiple views of your web site. New browser windows are also useful for temporary display of generated HTML, and, when created without the menu bar and other user controls, these windows can serve as dialog boxes for more complex messages or user input.

JavaScript does not define methods that allow you to create and manipulate frames directly within a browser window. However, the ability to generate HTML dynamically allows you to programmatically write the HTML tags that create any desired frame layout.

JavaScript also allows control over which web pages are displayed in the browser. The Location object allows you to download and display the contents of any URL in any window or frame of the browser. The History object allows you to move forward and back within the user's browsing history, simulating the action of the browser's **Forward** and **Back** buttons.

Yet another method of the Window object allows JavaScript to display arbitrary messages to the user in the status line of any browser window.

1.6.3 Interact with HTML Forms

Another important aspect of client-side JavaScript is its ability to interact with HTML forms. This capability is provided by the Form object and the form element objects it can contain: Button, Checkbox, Hidden, Password, Radio, Reset, Select, Submit, Text, and Textarea objects. These element objects allow you to read and write the values of the input elements in the forms in a document. For example, an online catalog might use an HTML form to allow the user to enter his order and could use JavaScript to read the input from that form in order to compute the cost of the order, the sales tax, and the shipping charge. JavaScript programs like this are, in fact, very common on the Web. We'll see a program shortly that uses an HTML form and JavaScript to allow the user to compute monthly payments on a home mortgage or other loan. JavaScript has an obvious advantage over server-based scripts for applications like these: JavaScript code is executed on the client, so the form's contents don't have to be sent to the server in order for relatively simple computations to be performed.

Another common use of client-side JavaScript with forms is for validating form data before it is submitted. If client-side JavaScript is able to perform all necessary error checking of a user's input, no round trip to the server is required to detect and inform the user of trivial input errors. Client-side JavaScript can also perform preprocessing of input data, which can reduce the amount of data that must be transmitted to the server. In some cases, client-side JavaScript can eliminate the need for scripts on the server altogether! (On the other hand, JavaScript and server-side scripting do work well together. For example, a server-side program can dynamically create JavaScript code on the fly, just as it dynamically creates HTML.)

1.6.4 Interact with the User

An important feature of JavaScript is the ability to define event handlers -- arbitrary pieces of code to be executed when a particular event occurs. Usually, these events are initiated by the user, when, for example, she moves the mouse over a hypertext link, enters a value in a form, or clicks the **Submit** button in a form. This event-handling capability is a crucial one, because programming with graphical interfaces, such as HTML forms, inherently requires an event-driven model. JavaScript can trigger any kind of action in response to user events. Typical examples might be to display a special message in the status line when the user positions the mouse over a hypertext link or to pop up a confirmation dialog box when the user submits an important form.

1.6.5 Read and Write Client State with Cookies

A *cookie* is a small amount of state data stored permanently or temporarily by the client. Cookies may be transmitted along with a web page by the server to the client, which stores them locally. When the client later requests the same or a related web page, it passes the relevant cookies back to the server, which can use their values to alter the content it sends back to the client. Cookies allow a web page or web site to remember things about the client -- for example, that the user has previously visited the site, has already registered and obtained a password, or has expressed a preference about the color and layout of web pages. Cookies help you provide the state information that is missing from the stateless HTTP protocol of the Web.

When cookies were invented, they were intended for use exclusively by server-side scripts; although stored on the client, they could be read or written only by the server. JavaScript changed this, because JavaScript programs can read and write cookie values and can dynamically generate document content based on the value of cookies.

1.6.6 Still More Features

In addition to the features I have already discussed, JavaScript has many other capabilities, including the following:

- JavaScript can change the image displayed by an `` tag to produce image rollover and animation effects.
- JavaScript can interact with Java applets and other embedded objects that appear in the browser. JavaScript code can read and write the properties of these applets and objects and can also invoke any methods they define. This feature truly allows JavaScript to script Java.
- As mentioned at the start of this section, JavaScript can perform arbitrary computation. JavaScript has a floating-point data type, arithmetic operators that work with it, and a full complement of standard floating-point mathematical functions.
- The JavaScript Date object simplifies the process of computing and working with dates and times.
- The Document object supports a property that specifies the last-modified date for the current document. You can use it to automatically display a timestamp on any document.
- JavaScript has a `window.setTimeout()` method that allows a block of arbitrary JavaScript code to be executed some number of milliseconds in the future. This is useful for building delays or repetitive actions into a JavaScript program. In JavaScript 1.2, `setTimeout()` is augmented by another useful method called `setInterval()`.
- The Navigator object (named after the Netscape web browser, of course) has variables that specify the name and version of the browser that is running, as well as variables that identify the platform on which it is running. These variables allow scripts to customize their behavior based on browser or platform, so that

they can take advantage of extra capabilities supported by some versions or work around bugs that exist on some platforms.

- In client-side JavaScript 1.2, the Screen object provides information about the size and color-depth of the monitor on which the web browser is being displayed.
- As of JavaScript 1.1, the `scroll()` method of the Window object allows JavaScript programs to scroll windows in the X and Y dimensions. In JavaScript 1.2, this method is augmented by a host of others that allow browser windows to be moved and resized.

1.6.7 What JavaScript Can't Do

Client-side JavaScript has an impressive list of capabilities. Note, however, that they are confined to browser- and document-related tasks. Since client-side JavaScript is used in a limited context, it does not have features that would be required for standalone languages:

- JavaScript does not have any graphics capabilities, except for the powerful ability to dynamically generate HTML (including images, tables, frames, forms, fonts, etc.) for the browser to display.
- For security reasons, client-side JavaScript does not allow the reading or writing of files. Obviously, you wouldn't want to allow an untrusted program from any random web site to run on your computer and rearrange your files!
- JavaScript does not support networking of any kind, except that it can cause the browser to download arbitrary URLs and it can send the contents of HTML forms across the network to server-side scripts and email addresses.

1.7 JavaScript Security

Any time that programs (such as JavaScript scripts, Visual Basic programs, or Microsoft Word macros) are included within shared documents, particularly documents that are transmitted over the Internet or by email, there is a potential for viruses or other malicious programs. The designers of JavaScript were aware of these security issues and took care not to give JavaScript programs the power to perform damaging acts. As described previously, for example, client-side JavaScript programs cannot read local files or perform networking operations.

Because of the complexity of the web-browser environment, however, a number of security problems did arise in early browser versions. In Netscape 2, for example, it was possible to write JavaScript code that could automatically steal the email address of any visitor to a page containing the code and then automatically send email in the visitor's name, without the visitor's knowledge or approval. This, and a number of other security holes, have been fixed. Although there is no guarantee that other security holes will not be found, most knowledgeable users are comfortable letting modern browsers run the JavaScript code found in web pages. [Chapter 21](#) contains a complete discussion of security in client-side JavaScript.

1.8 Example: Computing Loan Payments with JavaScript

[Example 1-3](#) is a listing of a complete, nontrivial JavaScript program. The program computes the monthly payment on a home mortgage or other loan, given the amount of the loan, the interest rate, and the repayment period. As you can see, the program consists of an HTML form made interactive with JavaScript code. [Figure 1-3](#) shows what the HTML form looks like when displayed in a web browser. But the figure can only capture a static snapshot of the program. The addition of JavaScript code makes it dynamic: whenever the user changes the amount of the loan, the interest rate, or the number of payments, the JavaScript code recomputes the monthly payment, the total of all payments, and the total interest paid over the lifetime of the loan.

Figure 1-3. A JavaScript loan payment calculator

Enter Loan Information:	
1) Amount of the loan (any currency):	<input type="text" value="200000"/>
2) Annual percentage rate of interest:	<input type="text" value="6.5"/>
3) Repayment period in years:	<input type="text" value="30"/>
<input type="button" value="Compute"/>	
Payment Information:	
4) Your monthly payment will be:	<input type="text" value="1264.14"/>
5) Your total payment will be:	<input type="text" value="455088.98"/>
6) Your total interest payments will be:	<input type="text" value="255088.98"/>

The first half of the example is an HTML form, nicely formatted using an HTML table. Note that several of the form elements define `onchange` or `onclick` event handlers. The web browser triggers these event handlers when the user changes the input or clicks on the **Compute** button displayed in the form. Note that in each case, the value of the event handler attribute is a string of JavaScript code: `calculate()`. When the event handler is triggered, it executes this code, which causes it to call the function `calculate()`.

The `calculate()` function is defined in the second half of the example, inside `<script>` tags. The function reads the user's input from the form, does the math required to compute the loan payments, and displays the results of these calculations using the bottom three form elements.

[Example 1-3](#) is simple, but it is worth taking the time to look at it carefully. You shouldn't expect to understand all the JavaScript code at this point, but studying this example should give you a good idea of what JavaScript programs look like, how event handlers work, and how JavaScript code can be integrated with HTML forms. Note that comments (in English) are included within HTML between `<!--` and `-->` markers and within JavaScript code in lines that begin with the characters `//`.

Example 1-3. Computing loan payments with JavaScript

```
<head><title>JavaScript Loan Calculator</title></head>
<body bgcolor="white">
<!--
  This is an HTML form that allows the user to enter data and allows
  JavaScript to display the results it computes back to the user. The
  form elements are embedded in a table to improve their appearance.
  The form itself is given the name "loandata", and the fields within
  the form are given names such as "interest" and "years". These
  field names are used in the JavaScript code that follows the form.
  Note that some of the form elements define "onchange" or "onclick"
  event handlers. These specify strings of JavaScript code to be
  executed when the user enters data or clicks on a button.
-->
<form name="loandata">
  <table>
    <tr><td colspan="3"><b>Enter Loan Information:</b></td></tr>
    <tr>
      <td>1)</td>
      <td>Amount of the loan (any currency):</td>
      <td><input type="text" name="principal" size="12"
        onchange="calculate();"></td>
    </tr>
    <tr>
      <td>2)</td>
      <td>Annual percentage rate of interest:</td>
      <td><input type="text" name="interest" size="12"
        onchange="calculate();"></td>
    </tr>
    <tr>
      <td>3)</td>
      <td>Repayment period in years:</td>
      <td><input type="text" name="years" size="12"
        onchange="calculate();"></td>
    </tr>
    <tr><td colspan="3">
      <input type="button" value="Compute" onclick="calculate();">
    </td></tr>
    <tr><td colspan="3">
      <b>Payment Information:</b>
    </td></tr>
    <tr>
      <td>4)</td>
      <td>Your monthly payment will be:</td>
      <td><input type="text" name="payment" size="12"></td>
    </tr>
    <tr>
      <td>5)</td>
```

```

        <td>Your total payment will be:</td>
        <td><input type="text" name="total" size="12"></td>
    </tr>
    <tr>
        <td>6)</td>
        <td>Your total interest payments will be:</td>
        <td><input type="text" name="totalinterest" size="12"></td>
    </tr>
</table>
</form>

```

```
<!--
```

This is the JavaScript program that makes the example work. Note that this script defines the calculate() function called by the event handlers in the form. The function refers to values in the form fields using the names defined in the HTML code above.

```
-->
```

```

<script language="JavaScript">
function calculate() {
    // Get the user's input from the form. Assume it is all valid.
    // Convert interest from a percentage to a decimal, and convert
from
    // an annual rate to a monthly rate. Convert payment period in
years
    // to the number of monthly payments.
    var principal = document.loandata.principal.value;
    var interest = document.loandata.interest.value / 100 / 12;
    var payments = document.loandata.years.value * 12;

    // Now compute the monthly payment figure, using esoteric math.
    var x = Math.pow(1 + interest, payments);
    var monthly = (principal*x*interest)/(x-1);

    // Check that the result is a finite number. If so, display the
results.
    if (!isNaN(monthly) &&
        (monthly != Number.POSITIVE_INFINITY) &&
        (monthly != Number.NEGATIVE_INFINITY)) {

        document.loandata.payment.value = round(monthly);
        document.loandata.total.value = round(monthly * payments);
        document.loandata.totalinterest.value =
            round((monthly * payments) - principal);
    }
    // Otherwise, the user's input was probably invalid, so don't
// display anything.
    else {
        document.loandata.payment.value = "";
        document.loandata.total.value = "";
        document.loandata.totalinterest.value = "";
    }
}

// This simple method rounds a number to two decimal places.
function round(x) {
    return Math.round(x*100)/100;
}

```

```
</script>  
</body>  
</html>
```

1.9 Using the Rest of This Book

The rest of this book is in five parts. [Part I](#), which immediately follows this chapter, documents the core JavaScript language. [Chapter 2](#) through [Chapter 6](#) begin this section with some bland but necessary reading -- these chapters cover the basic information you need to understand when learning a new programming language:

- [Chapter 2](#) explains the basic structure of the language.
- [Chapter 3](#) documents the data types supported by JavaScript.
- [Chapter 4](#) covers variables, variable scope, and related topics.
- [Chapter 5](#) explains expressions in JavaScript and documents each of the operators supported by JavaScript. Because JavaScript syntax is modeled on Java, which is, in turn, modeled on C and C++, experienced C, C++, or Java programmers can skim much of this chapter.
- [Chapter 6](#) describes the syntax and usage of each of the JavaScript statements. Again, experienced C, C++, and Java programmers can skim some, but not all, of this chapter.

The next five chapters of this first section become more interesting. They still cover the core of the JavaScript language, but they document parts of the language that will not already be familiar to you even if you already know C or Java. These chapters must be studied carefully if you want to really understand JavaScript:

- [Chapter 7](#) documents how functions are defined, invoked, and manipulated in JavaScript.
- [Chapter 8](#) explains objects, the most important JavaScript data type. This chapter discusses object-oriented programming in JavaScript and explains how you can define your own classes of objects in JavaScript.
- [Chapter 9](#) describes the creation and use of arrays in JavaScript.
- [Chapter 10](#) explains how to use regular expressions in JavaScript to perform pattern-matching and search-and-replace operations.
- [Chapter 11](#) covers advanced topics that have not been covered elsewhere. You can skip this chapter the first time through the book, but the material it contains is important to understand if you want to become a JavaScript expert.

[Part II](#) explains client-side JavaScript. The chapters in this part document the web-browser objects that are at the heart of client-side JavaScript and provide detailed examples of their use. Any interesting JavaScript program running in a web browser will rely heavily on features specific to the client side.

Here's what you'll find in [Part II](#):

- [Chapter 12](#) explains the integration of JavaScript with web browsers. It discusses the web browser as a programming environment and explains the various ways in which JavaScript is integrated into web pages for execution on the client side.
- [Chapter 13](#) documents the most central and important object of client-side JavaScript, the Window object, as well as several important window-related objects.
- [Chapter 14](#) explains the Document object and related objects that expose the contents of an HTML document to JavaScript code.
- [Chapter 15](#) documents the Form object, which represents HTML forms. It also documents the various form element objects that appear within HTML forms and shows examples of JavaScript programming using forms.
- [Chapter 16](#) illustrates the use of cookies to save state in web programming.
- [Chapter 17](#) explains the core pieces of the W3C DOM standard and shows how a JavaScript script can access any element of an HTML document.
- [Chapter 18](#) explains the portions of the W3C DOM standard that allow a JavaScript program to manipulate the style, appearance, and position of the elements within an HTML document. This chapter shows how you can create many DHTML effects with CSS properties.
- [Chapter 19](#) covers JavaScript events and event handlers, which are central to all JavaScript programs that interact with the user. This chapter covers the traditional event model, the W3C DOM standard event model, and the Internet Explorer proprietary event model.
- [Chapter 20](#) explores the important issue of compatibility in JavaScript programming and discusses techniques you can use to write JavaScript programs that run correctly (or fail gracefully) on a wide variety of web browsers.
- [Chapter 21](#) enumerates the security restrictions built into client-side JavaScript and explains the rationale for them.
- [Chapter 22](#) explains how you can use JavaScript to communicate with and control Java applets. It also covers how you can do the reverse -- invoke JavaScript code from Java applets.

[Part III](#), [Part IV](#), and [Part V](#) are reference sections that document the objects defined by the core JavaScript language, the objects defined in traditional client-side JavaScript programming, and the objects defined by the new W3C DOM standard, respectively.

1.10 Exploring JavaScript

The way to really learn a new programming language is to write programs with it. As you read through this book, I encourage you to try out JavaScript features as you learn about them. There are a number of techniques that make it easy to experiment with JavaScript.

The most obvious way to explore JavaScript is to write simple scripts. One of the nice things about client-side JavaScript is that anyone with a web browser and a simple text editor has a complete development environment; there is no need to buy or download special-purpose software in order to begin writing JavaScript scripts. We saw an example

that computed factorials at the beginning of this chapter. Suppose you wanted to modify it as follows to display Fibonacci numbers instead:

```
<script>
document.write("<h2>Table of Fibonacci Numbers</h2>");
for (i=0, j=1, k=0, fib =0; i<50; i++, fib=j+k, j=k, k=fib){
    document.write("Fibonacci (" + i + ") = " + fib);
    document.write("<br>");
}
</script>
```

This code may be convoluted (and don't worry if you don't yet understand it), but the point is that when you want to experiment with short programs like this, you can simply type them up and try them out in your web browser using a local `file:` URL. Note that the code uses the `document.write()` method to display its HTML output, so that you can see the results of its computations. This is an important technique for experimenting with JavaScript. As an alternative, you can also use the `alert()` method to display plain-text output in a dialog box:

```
alert("Fibonacci (" + i + ") = " + fib);
```

Note also that for simple JavaScript experiments like this, you can usually omit the `<html>`, `<head>`, and `<body>` tags in your HTML file.

For even simpler experiments with JavaScript, you can sometimes use the `javascript:` URL pseudoprotocol to evaluate a JavaScript expression and return the result. A JavaScript URL consists of the `javascript:` protocol specifier followed by arbitrary JavaScript code (with statements separated from one another by semicolons). When the browser loads such a URL, it executes the JavaScript code. The value of the last expression in such a URL is converted to a string, and this string is displayed by the web browser as its new document. For example, you might type the following JavaScript URLs into the **Location** field of your web browser to test your understanding of some of JavaScript's operators and statements:

```
javascript:5%2
javascript:x = 3; (x < 5)? "x is less": "x is greater"
javascript:d = new Date( ); typeof d;
javascript:for(i=0,j=1,k=0,fib=1; i<10; i++,fib=j+k,k=j,j=fib)
alert(fib);
javascript:s=""; for(i in document) s+=i+": "+document[i]+"\\n";
alert(s);
```

While exploring JavaScript, you'll probably write code that doesn't work as you expect it to and want to debug it. The basic debugging technique for JavaScript is like that in many other languages: insert statements into your code to print out the values of relevant variables so that you can try to figure out what is actually happening. As we've seen, you

can sometimes use the `document.write()` method to do this. This method doesn't work from within event handlers, however, and has some other shortcomings as well, so it's often easier to use the `alert()` function to display debugging messages in a separate dialog box.

The `for/in` loop (described in [Chapter 6](#)) is also useful for debugging. You can use it, along with the `alert()` method, to write a function that displays a list of the names and values of all properties of an object, for example. This kind of function can be handy when exploring the language or trying to debug code.

Good luck with JavaScript, and have fun exploring!

Part I: Core JavaScript

This part of the book, [Chapter 2](#) through [Chapter 11](#), documents the core JavaScript language as it is used in web browsers, web servers, and other embedded JavaScript implementations. This part is meant to be a JavaScript language reference. After you read through it once to learn the language, you may find yourself referring back to it to refresh your memory about some of the trickier points of JavaScript.

- [Chapter 2](#)
- [Chapter 3](#)
- [Chapter 4](#)
- [Chapter 5](#)
- [Chapter 6](#)
- [Chapter 7](#)
- [Chapter 8](#)
- [Chapter 9](#)
- [Chapter 10](#)
- [Chapter 11](#)

Chapter 2. Lexical Structure

The lexical structure of a programming language is the set of elementary rules that specifies how you write programs in that language. It is the lowest-level syntax of a language; it specifies such things as what variable names look like, what characters are used for comments, and how one program statement is separated from the next. This short chapter documents the lexical structure of JavaScript.

2.1 Character Set

JavaScript programs are written using the Unicode character set. Unlike the 7-bit ASCII encoding, which is useful only for English, and the 8-bit ISO Latin-1 encoding, which is useful only for English and major Western European languages, the 16-bit Unicode encoding can represent virtually every written language in common use on the planet. This is an important feature for internationalization and is particularly important for programmers who do not speak English.

American and other English-speaking programmers typically write programs using a text editor that supports only the ASCII or Latin-1 character encodings, and thus they don't have easy access to the full Unicode character set. This is not a problem, however, because both the ASCII and Latin-1 encodings are subsets of Unicode, so any JavaScript program written using those character sets is perfectly valid. Programmers who are used to thinking of characters as 8-bit quantities may be disconcerted to know that JavaScript represents each character using 2 bytes, but this fact is actually transparent to the programmer and can simply be ignored.

Although the ECMAScript v3 standard allows Unicode characters anywhere in a JavaScript program, Versions 1 and 2 of the standard allow Unicode characters only in comments and quoted string literals -- all other parts of an ECMAScript v1 program are restricted to the ASCII character set. Versions of JavaScript that predate ECMAScript standardization typically do not support Unicode at all.

2.2 Case Sensitivity

JavaScript is a case-sensitive language. This means that language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters. The `while` keyword, for example, must be typed "while", not "While" or "WHILE". Similarly, `online`, `Online`, `OnLine`, and `ONLINE` are four distinct variable names.

Note, however, that HTML is not case-sensitive. Because of its close association with client-side JavaScript, this difference can be confusing. Many JavaScript objects and properties have the same names as the HTML tags and attributes they represent. While these tags and attribute names can be typed in any case in HTML, in JavaScript they

typically must be all lowercase. For example, the HTML `onclick` event handler attribute is commonly specified as `onClick` in HTML, but it must be referred to as `onclick` in JavaScript code.

While core JavaScript is entirely and exclusively case-sensitive, exceptions to this rule are allowed in client-side JavaScript. In Internet Explorer 3, for example, all client-side objects and properties were case-insensitive. This caused problematic incompatibilities with Netscape, however, so in Internet Explorer 4 and later, client-side objects and properties are case-sensitive.

2.3 Whitespace and Line Breaks

JavaScript ignores spaces, tabs, and newlines that appear between tokens in programs, except those that are part of string or regular expression literals. A *token* is a keyword, variable name, number, function name, or some other entity in which you would obviously not want to insert a space or a line break. If you place a space, tab, or newline within a token, you break it up into two tokens -- thus, `123` is a single numeric token, but `12 3` is two separate tokens (and constitutes a syntax error, incidentally).

Because you can use spaces, tabs, and newlines freely in your programs (except in strings, regular expressions, and tokens), you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand. Note, however, that there is one minor restriction on the placement of line breaks; it is described in the following section.

2.4 Optional Semicolons

Simple statements in JavaScript are generally followed by semicolons (`;`), just as they are in C, C++, and Java. The semicolon serves to separate statements from each other. In JavaScript, however, you may omit the semicolon if each of your statements is placed on a separate line. For example, the following code could be written without semicolons:

```
a = 3;  
b = 4;
```

But when formatted as follows, the first semicolon is required:

```
a = 3; b = 4;
```

Omitting semicolons is not a good programming practice; you should get in the habit of using them.

Although JavaScript theoretically allows line breaks between any two tokens, the fact that JavaScript automatically inserts semicolons for you causes some exceptions to this

rule. Loosely, if you break a line of code in such a way that the line before the break appears to be a complete statement, JavaScript may think you omitted the semicolon and insert one for you, altering your meaning. Some places you should look out for this are with the `return`, `break`, and `continue` statements (which are described in [Chapter 6](#)). For example, consider the following:

```
return  
true;
```

JavaScript assumes you meant:

```
return;  
true;
```

However, you probably meant:

```
return true;
```

This is something to watch out for -- this code does not cause a syntax error and will fail in a nonobvious way. A similar problem occurs if you write:

```
break  
outerloop;
```

JavaScript inserts a semicolon after the `break` keyword, causing a syntax error when it tries to interpret the next line. For similar reasons, the `++` and `--` postfix operators (see [Chapter 5](#)) must always appear on the same line as the expressions to which they are applied.

2.5 Comments

JavaScript, like Java, supports both C++ and C-style comments. Any text between a `//` and the end of a line is treated as a comment and is ignored by JavaScript. Any text between the characters `/*` and `*/` is also treated as a comment. These C-style comments may span multiple lines but may not be nested. The following lines of code are all legal JavaScript comments:

```
// This is a single-line comment.  
/* This is also a comment */ // and here is another comment.  
/*  
 * This is yet another comment.  
 * It has multiple lines.  
 */
```

2.6 Literals

A *literal* is a data value that appears directly in a program. The following are all literals:

```
12           // The number twelve
1.2         // The number one point two
"hello world" // A string of text
'Hi'        // Another string
true        // A boolean value
false       // The other boolean value
/javascript/gi // A "regular expression" literal (for pattern
matching)
null        // Absence of an object
```

In ECMAScript v3, expressions that serve as array and object literals are also supported. For example:

```
{ x:1, y:2 } // An object initializer
[1,2,3,4,5]  // An array initializer
```

Note that these array and object literals have been supported since JavaScript 1.2 but were not standardized until ECMAScript v3.

Literals are an important part of any programming language, as it is impossible to write a program without them. The various JavaScript literals are described in detail in [Chapter 3](#).

2.7 Identifiers

An *identifier* is simply a name. In JavaScript, identifiers are used to name variables and functions and to provide labels for certain loops in JavaScript code. The rules for legal identifier names are the same in JavaScript as they are in Java and many other languages. The first character must be a letter, an underscore (`_`), or a dollar sign (`$`).^[1] Subsequent characters may be any letter or digit or an underscore or dollar sign. (Numbers are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.) These are all legal identifiers:

^[1] Note that dollar signs are not legal in identifiers prior to JavaScript 1.1. They are intended for use only by code-generation tools, so you should avoid using dollar signs in identifiers in the code you write yourself.

```
i
my_variable_name
v13
_dummy
$str
```

In ECMAScript v3, identifiers can contain letters and digits from the complete Unicode character set. Prior to this version of the standard, JavaScript identifiers are restricted to the ASCII character set. ECMAScript v3 also allows Unicode escape sequences to appear in identifiers. A Unicode escape is the characters `\u` followed by 4 hexadecimal digits that specify a 16-bit character encoding. For example, the identifier `π` could also be written as `\u03c0`. Although this is an awkward syntax, it makes it possible to translate JavaScript programs that contain Unicode characters into a form that allows them to be manipulated with text editors and other tools that do not support the full Unicode character set.

Finally, identifiers cannot be the same as any of the keywords used for other purposes in JavaScript. The next section lists the special names that are reserved in JavaScript.

2.8 Reserved Words

There are a number of reserved words in JavaScript. These are words that you cannot use as identifiers (variable names, function names, and loop labels) in your JavaScript programs. [Table 2-1](#) lists the keywords standardized by ECMAScript v3. These words have special meaning to JavaScript -- they are part of the language syntax itself.

<code>break</code>	<code>do</code>	<code>if</code>	<code>switch</code>	<code>typeof</code>
<code>case</code>	<code>else</code>	<code>in</code>	<code>this</code>	<code>var</code>
<code>catch</code>	<code>false</code>	<code>instanceof</code>	<code>throw</code>	<code>void</code>
<code>continue</code>	<code>finally</code>	<code>new</code>	<code>true</code>	<code>while</code>
<code>default</code>	<code>for</code>	<code>null</code>	<code>try</code>	<code>with</code>
<code>delete</code>	<code>function</code>	<code>return</code>		

[Table 2-2](#) lists other reserved keywords. These words are not currently used in JavaScript, but they are reserved by ECMAScript v3 as possible future extensions to the language.

<code>abstract</code>	<code>double</code>	<code>goto</code>	<code>native</code>	<code>static</code>
<code>boolean</code>	<code>enum</code>	<code>implements</code>	<code>package</code>	<code>super</code>
<code>byte</code>	<code>export</code>	<code>import</code>	<code>private</code>	<code>synchronized</code>
<code>char</code>	<code>extends</code>	<code>int</code>	<code>protected</code>	<code>throws</code>

Table 2-2. Words reserved for ECMA extensions

abstract	double	goto	native	static
class	final	interface	public	transient
const	float	long	short	volatile
debugger				

In addition to some of the formally reserved words just listed, current drafts of the ECMAScript v4 standard are contemplating the use of the keywords `as`, `is`, `namespace`, and `use`. Current JavaScript interpreters will not prevent you from using these four words as identifiers, but you should avoid them anyway.

You should also avoid using as identifiers the names of global variables and functions that are predefined by JavaScript. If you create variables or functions with these names, either you will get an error (if the property is read-only) or you will redefine the existing variable or function -- something you should not do unless you know exactly what you're doing. [Table 2-3](#) lists global variables and functions defined by the ECMAScript v3 standard. Specific implementations may define other global properties, and each specific JavaScript embedding (client-side, server-side, etc.) will have its own extensive list of global properties.^[2]

^[2] See the Window object in the client-side reference section of this book for a list of the additional global variables and functions defined by client-side JavaScript.

Table 2-3. Other identifiers to avoid

arguments	encodeURIComponent	Infinity	Object	String
Array	Error	isFinite	parseFloat	SyntaxError
Boolean	escape	isNaN	parseInt	TypeError
Date	eval	Math	RangeError	undefined
decodeURI	EvalError	NaN	ReferenceError	unescape
decodeURIComponent	Function	Number	RegExp	URIError

Chapter 3. Data Types and Values

Computer programs work by manipulating *values*, such as the number 3.14 or the text "Hello World". The types of values that can be represented and manipulated in a programming language are known as *data types*, and one of the most fundamental characteristics of a programming language is the set of data types it supports. JavaScript allows you to work with three primitive data types: numbers, strings of text (known as "strings"), and boolean truth values (known as "booleans"). JavaScript also defines two trivial data types, null and undefined, each of which defines only a single value.

In addition to these primitive data types, JavaScript supports a composite data type known as object. An object (that is, a member of the data type object) represents a collection of values (either primitive values, like numbers and strings, or composite values, like other objects). Objects in JavaScript have a dual nature: an object can represent an unordered collection of named values or an ordered collection of numbered values. In the latter case, the object is called an *array*. Although objects and arrays are fundamentally the same data type in JavaScript, they behave quite differently and will usually be considered distinct types throughout this book.

JavaScript defines another special kind of object, known as a *function*. A function is an object that has executable code associated with it. A function may be *invoked* to perform some kind of operation. Like arrays, functions behave differently from other kinds of objects, and JavaScript defines special language syntax for working with them. Thus, we'll treat the function data type independently of the object and array types.

In addition to functions and arrays, core JavaScript defines a few other specialized kinds of objects. These objects do not represent new data types, just new *classes* of objects. The Date class defines objects that represent dates, the RegExp class defines objects that represent regular expressions (a powerful pattern-matching tool described in [Chapter 10](#)), and the Error class defines objects that represent syntax and runtime errors that can occur in a JavaScript program.

The remainder of this chapter documents each of the primitive data types in detail. It also introduces the object, array, and function data types, which are fully documented in [Chapter 7](#), [Chapter 8](#), and [Chapter 9](#). Finally, it provides an overview of the Date, RegExp, and Error classes, which are documented in full detail in the core reference section of this book.

3.1 Numbers

Numbers are the most basic data type; they require very little explanation. JavaScript differs from programming languages such as C and Java in that it does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard,^[1] which means it can represent

numbers as large as $\pm 1.7976931348623157 \times 10^{308}$ and as small as $\pm 5 \times 10^{-324}$.

^[1] This format should be familiar to Java programmers as the format of the `double` type. It is also the `double` format used in almost all modern implementations of C and C++.

When a number appears directly in a JavaScript program, we call it a numeric literal. JavaScript supports numeric literals in several formats, as described in the following sections. Note that any numeric literal can be preceded by a minus sign (-) to make the number negative. Technically, however, - is the unary negation operator (see [Chapter 5](#)), not part of the numeric literal syntax.

3.1.1 Integer Literals

In a JavaScript program, a base-10 integer is written as a sequence of digits. For example:

```
0
3
10000000
```

The JavaScript number format allows you to exactly represent all integers between -9007199254740992 (-2^{53}) and 9007199254740992 (253), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits. Note, however, that certain integer operations in JavaScript (in particular the bitwise operators described in [Chapter 5](#)) are performed on 32-bit integers, which range from -2147483648 (-2^{31}) to 2147483647 ($2^{31} - 1$).

3.1.2 Hexadecimal and Octal Literals

In addition to base-10 integer literals, JavaScript recognizes hexadecimal (base-16) values. A hexadecimal literal begins with "0x" or "0X", followed by a string of hexadecimal digits. A hexadecimal digit is one of the digits 0 through 9 or the letters a (or A) through f (or F), which are used to represent values 10 through 15. Examples of hexadecimal integer literals are:

```
0xff // 15*16 + 15 = 255 (base 10)
0xCAFE911
```

Although the ECMAScript standard does not support them, some implementations of JavaScript allow you to specify integer literals in octal (base-8) format. An octal literal begins with the digit 0 and is followed by a sequence of digits, each between 0 and 7. For example:

```
0377 // 3*64 + 7*8 + 7 = 255 (base 10)
```

write an integer literal with a leading zero -- you cannot know whether an implementation will interpret it as an octal or decimal value.

3.1.3 Floating-Point Literals

Floating-point literals can have a decimal point; they use the traditional syntax for real numbers. A real value is represented as the integral part of the number, followed by a decimal point and the fractional part of the number.

Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent. This notation represents the real number multiplied by 10 to the power of the exponent.

More succinctly, the syntax is:

```
[digits][.digits][(E|e)[(+|-)]digits]
```

For example:

```
3.14
2345.789
.33333333333333333333
6.02e23 // 6.02 x 1023
1.4738223E-32 // 1.4738223 x 10-32
```

Note that there are infinitely many real numbers, but only a finite number of them (18437736874454810627, to be exact) can be represented exactly by the JavaScript floating-point format. This means that when you're working with real numbers in JavaScript, the representation of the number will often be an approximation of the actual number. The approximation is usually good enough, however, and this is rarely a practical problem.

3.1.4 Working with Numbers

JavaScript programs work with numbers using the arithmetic operators that the language provides. These include + for addition, - for subtraction, * for multiplication, and / for division. Full details on these and other arithmetic operators can be found in [Chapter 5](#).

In addition to these basic arithmetic operations, JavaScript supports more complex mathematical operations through a large number of mathematical functions that are a core part of the language. For convenience, these functions are all stored as properties of a single `Math` object, so we always use the literal name `Math` to access them. For example, here's how to compute the sine of the numeric value `x`:

```
sine_of_x = Math.sin(x);
```

And to compute the square root of a numeric expression:

```
hypot = Math.sqrt(x*x + y*y);
```

See the `Math` object and subsequent listings in the core reference section of this book for full details on all the mathematical functions supported by JavaScript.

There is also one interesting method that you can use with numbers. The `toString()` method converts an integer to a string, using the radix, or base, specified by its argument (the base must be between 2 and 36). For example, to convert a number to binary, use `toString()` like this:

```
var x = 33;  
var y = x.toString(2); // y is "100001"
```

To invoke the `toString()` method on a number literal, you can use parentheses to prevent the `.` from being interpreted as a decimal point:

```
var y = (257).toString(0x10); // y is "101"
```

3.1.5 Special Numeric Values

JavaScript uses several special numeric values. When a floating-point value becomes larger than the largest representable finite number, the result is a special infinity value, which JavaScript prints as `Infinity`. Similarly, when a negative value becomes lower than the last representable negative number, the result is negative infinity, printed as `-Infinity`.

Another special JavaScript numeric value is returned when a mathematical operation (such as division of zero by zero) yields an undefined result or an error. In this case, the result is the special not-a-number value, printed as `NaN`. The not-a-number value behaves unusually: it does not compare equal to any number, including itself! For this reason, a special function, `isNaN()`, is required to test for this value. A related function, `isFinite()`, tests whether a number is not NaN and is not positive or negative infinity.

[Table 3-1](#) lists several constants that JavaScript defines to represent these special numeric values.

<p>Table 3-1. Special numeric constants</p>
--

Constant	Meaning
<code>Infinity</code>	Special value to represent infinity
<code>NaN</code>	Special not-a-number value
<code>Number.MAX_VALUE</code>	Largest representable number
<code>Number.MIN_VALUE</code>	Smallest (closest to zero) representable number
<code>Number.NaN</code>	Special not-a-number value
<code>Number.POSITIVE_INFINITY</code>	Special value to represent infinity
<code>Number.NEGATIVE_INFINITY</code>	Special value to represent negative infinity

The `Infinity` and `NaN` constants are defined by the ECMAScript v1 standard and are not implemented prior to JavaScript 1.3. The various `Number` constants, however, have been implemented since JavaScript 1.1.

3.2 Strings

A *string* is a sequence of Unicode letters, digits, punctuation characters, and so on -- it is the JavaScript data type for representing text. As you'll see shortly, you can include string literals in your programs by enclosing them in matching pairs of single or double quotation marks. Note that JavaScript does not have a character data type such as `char`, like C, C++, and Java do. To represent a single character, you simply use a string that has a length of 1.

3.2.1 String Literals

A string is a sequence of zero or more Unicode characters enclosed within single or double quotes (`'` or `"`). Double-quote characters may be contained within strings delimited by single-quote characters, and single-quote characters may be contained within strings delimited by double quotes. String literals must be written on a single line; they may not be broken across two lines. If you need to include a newline character in a string literal, use the character sequence `\n` , which is documented in the next section. Examples of string literals are:

```

"" // The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\nhas two lines"
"pi is the ratio of a circle's circumference to its diameter"

```

As illustrated in the last example string shown, the ECMAScript v1 standard allows Unicode characters within string literals. Implementations prior to JavaScript 1.3, however, typically support only ASCII or Latin-1 characters in strings. As we'll see in the next section, you can also include Unicode characters in your string literals using special "escape sequences." This is useful if your text editor does not provide complete Unicode support.

Note that when you use single quotes to delimit your strings, you must be careful with English contractions and possessives like can't and O'Reilly's. Since the apostrophe is the same as the single-quote character, you must use the backslash character (\) to escape any apostrophes that appear in single-quoted strings (this is explained in the next section).

In client-side JavaScript programming, JavaScript code often contains strings of HTML code, and HTML code often contains strings of JavaScript code. Like JavaScript, HTML uses either single or double quotes to delimit its strings. Thus, when combining JavaScript and HTML, it is a good idea to use one style of quotes for JavaScript and the other style for HTML. In the following example, the string "Thank you" is single-quoted within a JavaScript expression, which is double-quoted within an HTML event handler attribute:

```
<a href="" onclick="alert('Thank you')">Click Me</a>
```

3.2.2 Escape Sequences in String Literals

The backslash character (\) has a special purpose in JavaScript strings. Combined with the character that follows it, it represents a character that is not otherwise representable within the string. For example, \n is an *escape sequence* that represents a newline character.^[2]

^[2] C, C++, and Java programmers will already be familiar with this and other JavaScript escape sequences.

Another example, mentioned in the previous section, is the \' escape, which represents the single quote (or apostrophe) character. This escape sequence is useful when you need to include an apostrophe in a string literal that is contained within single quotes. You can see why we call these escape sequences -- here, the backslash allows us to escape from the usual interpretation of the single-quote character. Instead of using it to mark the end of the string, we use it as an apostrophe:

```
'You\'re right, it can\'t be a quote'
```

[Table 3-2](#) lists the JavaScript escape sequences and the characters they represent. Two of the escape sequences are generic ones that can be used to represent any character by specifying its Latin-1 or Unicode character code as a hexadecimal number. For example,

the sequence `\xA9` represents the copyright symbol, which has the Latin-1 encoding given by the hexadecimal number A9. Similarly, the `\u` escape represents an arbitrary Unicode character specified by four hexadecimal digits. `\u03c0` represents the character π , for example. Note that Unicode escapes are required by the ECMAScript v1 standard but are not typically supported in implementations prior to JavaScript 1.3. Some implementations of JavaScript also allow a Latin-1 character to be specified by three octal digits following a backslash, but this escape sequence is not supported in the ECMAScript v3 standard and should no longer be used.

Table 3-2. JavaScript escape sequences

Sequence	Character represented
<code>\0</code>	The NUL character (<code>\u0000</code>).
<code>\b</code>	Backspace (<code>\u0008</code>).
<code>\t</code>	Horizontal tab (<code>\u0009</code>).
<code>\n</code>	Newline (<code>\u000A</code>).
<code>\v</code>	Vertical tab (<code>\u000B</code>).
<code>\f</code>	Form feed (<code>\u000C</code>).
<code>\r</code>	Carriage return (<code>\u000D</code>).
<code>\"</code>	Double quote (<code>\u0022</code>).
<code>\'</code>	Apostrophe or single quote (<code>\u0027</code>).
<code>\\</code>	Backslash (<code>\u005C</code>).
<code>\xxx</code>	The Latin-1 character specified by the two hexadecimal digits <code>xx</code> .
<code>\uXXXX</code>	The Unicode character specified by the four hexadecimal digits <code>XXXX</code> .
<code>\xxx</code>	The Latin-1 character specified by the octal digits <code>xxx</code> , between 1 and 377. Not supported by ECMAScript v3; do not use this escape sequence.

Finally, note that the backslash escape cannot be used before a line break to continue a string (or other JavaScript) token across two lines or to include a literal line break in a string. If the `\` character precedes any character other than those shown in [Table 3-2](#), the backslash is simply ignored (although future versions of the language may, of course, define new escape sequences). For example, `\#` is the same thing as `#`.

3.2.3 Working with Strings

One of the built-in features of JavaScript is the ability to *concatenate* strings. If you use the `+` operator with numbers, it adds them. But if you use this operator on strings, it joins them by appending the second to the first. For example:

```
msg = "Hello, " + "world";    // Produces the string "Hello, world"
greeting = "Welcome to my home page," + " " + name;
```

To determine the length of a string -- the number of characters it contains -- use the `length` property of the string. If the variable `s` contains a string, you access its length like this:

```
s.length
```

You can use a number of methods to operate on strings. For example, to get the last character of a string `s`:

```
last_char = s.charAt(s.length - 1)
```

To extract the second, third, and fourth characters from a string `s`:

```
sub = s.substring(1,4);
```

To find the position of the first letter `a` in a string `s`:

```
i = s.indexOf('a');
```

There are quite a few other methods that you can use to manipulate strings. You'll find full documentation of these methods in the core reference section of this book, under the `String` object and subsequent listings.

As you can tell from the previous examples, JavaScript strings (and JavaScript arrays, as we'll see later) are indexed starting with zero. That is, the first character in a string is character 0. C, C++, and Java programmers should be perfectly comfortable with this convention, but programmers used to languages with 1-based strings and arrays may find that it takes some getting used to.

In some implementations of JavaScript, individual characters can be read from strings (but not written into strings) using array notation, so the earlier call to `charAt()` could also be written like this:


```
last_char = s[s.length - 1];
```

Note, however, that this syntax is not part of the ECMAScript v3 standard, is not portable, and should be avoided.

When we discuss the object data type, you'll see that object properties and methods are used in the same way that string properties and methods are used in the previous examples. This does not mean that strings are a type of object. In fact, strings are a distinct JavaScript data type. They use object syntax for accessing properties and methods, but they are not themselves objects. We'll see just why this is at the end of this chapter.

3.3 Boolean Values

The number and string data types have a large or infinite number of possible values. The boolean data type, on the other hand, has only two. The two legal boolean values are represented by the literals `true` and `false`. A boolean value represents a truth value -- it says whether something is true or not.

Boolean values are generally the result of comparisons you make in your JavaScript programs. For example:

```
a == 4
```

This code tests to see if the value of the variable `a` is equal to the number `4`. If it is, the result of this comparison is the boolean value `true`. If `a` is not equal to `4`, the result of the comparison is `false`.

Boolean values are typically used in JavaScript control structures. For example, the `if/else` statement in JavaScript performs one action if a boolean value is `true` and another action if the value is `false`. You usually combine a comparison that creates a boolean value directly with a statement that uses it. The result looks like this:

```
if (a == 4)
  b = b + 1;
else
  a = a + 1;
```

This code checks if `a` equals `4`. If so, it adds `1` to `b`; otherwise, it adds `1` to `a`.

Instead of thinking of the two possible boolean values as `true` and `false`, it is sometimes convenient to think of them as `on` (`true`) and `off` (`false`) or `yes` (`true`) and `no` (`false`). Sometimes it is even useful to consider them equivalent to `1` (`true`) and `0` (`false`). (In fact, JavaScript does just this and converts `true` and `false` to `1` and `0` when necessary.)^[1]

^[3] C programmers should note that JavaScript has a distinct boolean data type, unlike C, which simply uses integer values to simulate boolean values. Java programmers should note that although JavaScript has a boolean type, it is not nearly as pure as the Java `boolean` data type -- JavaScript boolean values are easily converted to and from other data types, and so in practice, the use of boolean values in JavaScript is much more like their use in C than in Java.

3.4 Functions

A *function* is a piece of executable code that is defined by a JavaScript program or predefined by the JavaScript implementation. Although a function is defined only once, a JavaScript program can execute or invoke it any number of times. A function may be passed arguments, or parameters, specifying the value or values upon which it is to perform its computation, and it may also return a value that represents the results of that computation. JavaScript implementations provide many predefined functions, such as the `Math.sin()` function that computes the sine of an angle.

JavaScript programs may also define their own functions with code that looks like this:

```
function square(x) // The function is named square. It expects one
argument, x.
{
    // The body of the function begins here.
    return x*x; // The function squares its argument and returns
that value.
} // The function ends here.
```

Once a function is defined, you can invoke it by following the function's name with an optional comma-separated list of arguments within parentheses. The following lines are function invocations:

```
y = Math.sin(x);
y = square(x);
d = compute_distance(x1, y1, z1, x2, y2, z2);
move( );
```

An important feature of JavaScript is that functions are values that can be manipulated by JavaScript code. In many languages, including Java, functions are only a syntactic feature of the language -- they can be defined and invoked, but they are not data types. The fact that functions are true values in JavaScript gives a lot of flexibility to the language. It means that functions can be stored in variables, arrays, and objects, and it means that functions can be passed as arguments to other functions. This can quite often be useful. We'll learn more about defining and invoking functions, and also about using them as data values, in [Chapter 7](#).

Since functions are values just like numbers and strings, they can be assigned to object properties just like other values can. When a function is assigned to a property of an object (the object data type and object properties are described in [Section 3.5](#)), it is often referred to as a *method* of that object. Methods are an important part of object-oriented programming. We'll see more about them in [Chapter 8](#).

3.4.1 Function Literals

In the preceding section, we saw the definition of a function `square()`. The syntax shown there is used to define most functions in most JavaScript programs. However, ECMAScript v3 provides a syntax (implemented in JavaScript 1.2 and later) for defining function literals. A function literal is defined with the `function` keyword, followed by an optional function name, followed by a parenthesized list of function arguments and the body of the function within curly braces. In other words, a function literal looks just like a function definition, except that it does not have to have a name. The big difference is that function literals can appear within other JavaScript expressions. Thus, instead of defining the function `square()` with a function definition:

```
function square(x) { return x*x; }
```

We can define it with a function literal:

```
var square = function(x) { return x*x; }
```

Functions defined in this way are sometimes called *lambda functions* in homage to the LISP programming language, which was one of the first to allow unnamed functions to be embedded as literal data values within a program. Although it is not immediately obvious why one might choose to use function literals in a program, we'll see later that in advanced scripts they can be quite convenient and useful.

There is one other way to define a function: you can pass the argument list and the body of the function as strings to the `Function()` constructor. For example:

```
var square = new Function("x", "return x*x;");
```

Defining a function in this way is not often useful. It is usually awkward to express the function body as a string, and in many JavaScript implementations, functions defined in this way will be less efficient than functions defined in either of the other two ways.

3.5 Objects

An *object* is a collection of named values. These named values are usually referred to as *properties* of the object. (Sometimes they are called fields of the object, but this usage can be confusing.) To refer to a property of an object, you refer to the object, followed by a period and the name of the property. For example, if an object named `image` has properties named `width` and `height`, we can refer to those properties like this:

```
image.width  
image.height
```

Properties of objects are, in many ways, just like JavaScript variables; they can contain any type of data, including arrays, functions, and other objects. Thus, you might see JavaScript code like this:

```
document.myform.button
```

This code refers to the `button` property of an object that is itself stored in the `myform` property of an object named `document`.

As mentioned earlier, when a function value is stored in a property of an object, that function is often called a method, and the property name becomes the method name. To invoke a method of an object, use the `.` syntax to extract the function value from the object, and then use the `()` syntax to invoke that function. For example, to invoke the `write()` method of the Document object, you can use code like this:

```
document.write("this is a test");
```

Objects in JavaScript have the ability to serve as associative arrays -- that is, they can associate arbitrary data values with arbitrary strings. When an object is used in this way, a different syntax is generally required to access the object's properties: a string containing the name of the desired property is enclosed within square brackets. Using this syntax, we could access the properties of the `image` object mentioned previously with code like this:

```
image["width"]  
image["height"]
```

Associative arrays are a powerful data type; they are useful for a number of programming techniques. We'll learn more about objects in their traditional and associative array usages in [Chapter 8](#).

3.5.1 Creating Objects

As we'll see in [Chapter 8](#), objects are created by invoking special *constructor* functions. For example, the following lines all create new objects:

```
var o = new Object( );  
var now = new Date( );  
var pattern = new RegExp("\\sjava\\s", "i");
```

Once you have created an object of your own, you can use and set its properties however you desire:

```
var point = new Object( );
point.x = 2.3;
point.y = -1.2;
```

3.5.2 Object Literals

ECMAScript v3 defines (and JavaScript 1.2 implements) an object literal syntax that allows you to create an object and specify its properties. An object literal (also called an object initializer) consists of a comma-separated list of colon-separated property/value pairs, all enclosed within curly braces. Thus, the object `point` in the previous code could also be created and initialized with this line:

```
var point = { x:2.3, y:-1.2 };
```

Object literals can also be nested. For example:

```
var rectangle = { upperLeft: { x: 2, y: 2 },
                  lowerRight: { x: 4, y: 4 }
                };
```

Finally, the property values used in object literals need not be constant -- they can be arbitrary JavaScript expressions:

```
var square = { upperLeft: { x:point.x, y:point.y },
               lowerRight: { x:(point.x + side), y:(point.y+side) } };
```

3.6 Arrays

An *array* is a collection of data values, just as an object is. While each data value contained in an object has a name, each data value in an array has a number, or *index*. In JavaScript, you retrieve a value from an array by enclosing an index within square brackets after the array name. For example, if an array is named `a`, and `i` is a non-negative integer, `a[i]` is an element of the array. Array indexes begin with zero. Thus, `a[2]` refers to the *third* element of the array `a`.

Arrays may contain any type of JavaScript data, including references to other arrays or to objects or functions. For example:

```
document.images[1].width
```

This code refers to the `width` property of an object stored in the second element of an array stored in the `images` property of the `document` object.

Note that the arrays described here differ from the associative arrays described in [Section 3.5](#). The regular arrays we are discussing here are indexed by non-negative integers. Associative arrays are indexed by strings. Also note that JavaScript does not support multidimensional arrays, except as arrays of arrays. Finally, because JavaScript is an untyped language, the elements of an array do not all need to be of the same type, as they do in typed languages like Java. We'll learn more about arrays in [Chapter 9](#).

3.6.1 Creating Arrays

Arrays can be created with the `Array()` constructor function. Once created, any number of indexed elements can easily be assigned to the array:

```
var a = new Array( );
a[0] = 1.2;
a[1] = "JavaScript";
a[2] = true;
a[3] = { x:1, y:3 };
```

Arrays can also be initialized by passing array elements to the `Array()` constructor. Thus, the previous array-creation and -initialization code could also be written:

```
var a = new Array(1.2, "JavaScript", true, { x:1, y:3 });
```

If you pass only a single number to the `Array()` constructor, it specifies the length of the array. Thus:

```
var a = new Array(10);
```

creates a new array with 10 undefined elements.

3.6.2 Array Literals

ECMAScript v3 defines (and JavaScript 1.2 implements) a literal syntax for creating and initializing arrays. An array literal (or array initializer) is a comma-separated list of values contained within square brackets. The values within the brackets are assigned sequentially to array indexes starting with zero.^[4] For example, in JavaScript 1.2 the array creation and initialization code in the previous section could also be written as:

^[4] Netscape's JavaScript 1.2 implementation has a bug: when an array literal is specified with a number as its single element, that number specifies the length of the array rather than the value of the first element. While this behavior mirrors that of the `Array()` constructor, it is clearly inappropriate in this context.

```
var a = [1.2, "JavaScript", true, { x:1, y:3 }];
```

Like object literals, array literals can be nested:

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Also, as with object literals, the elements in array literals can be arbitrary expressions and need not be restricted to constants:

```
var base = 1024;  
var table = [base, base+1, base+2, base+3];
```

Undefined elements can be included in an array literal by simply omitting a value between commas. For example, the following array contains five elements, including three undefined elements:

```
var sparseArray = [1,,,5];
```

3.7 null

The JavaScript keyword `null` is a special value that indicates no value. `null` is usually considered to be a special value of object type -- a value that represents no object. `null` is a unique value, distinct from all other values. When a variable holds the value `null`, you know that it does not contain a valid object, array, number, string, or boolean value.^[5]

^[5] C and C++ programmers should note that `null` in JavaScript is not the same as `0`, as it is in those languages. In certain circumstances `null` is converted to `0`, but the two are not equivalent.

3.8 undefined

Another special value used occasionally by JavaScript is the `undefined` value returned when you use either a variable that has been declared but never had a value assigned to it, or an object property that does not exist. Note that this special `undefined` value is not the same as `null`.

Although `null` and the `undefined` value are distinct, the `==` equality operator considers them to be equal to one another. Consider the following:

```
my.prop == null
```

This comparison is `true` either if the `my.prop` property does not exist or if it does exist but contains the value `null`. Since both `null` and the `undefined` value indicate an

absence of value, this equality is often what we want. However, if you truly must distinguish between a `null` value and an `undefined` value, use the `===` identity operator or the `typeof` operator (see [Chapter 5](#) for details).

Unlike `null`, `undefined` is not a reserved word in JavaScript. The ECMAScript v3 standard specifies that there is always a global variable named `undefined` whose initial value is the `undefined` value. Thus, in a conforming implementation, you can treat `undefined` as a keyword, as long as you don't assign a value to the variable.

If you are not sure that your implementation has the `undefined` variable, you can simply declare your own:

```
var undefined;
```

By declaring but not initializing the variable, you assure that it has the `undefined` value. The `void` operator (see [Chapter 5](#)) provides another way to obtain the `undefined` value.

3.9 The Date Object

The previous sections have described all of the fundamental data types supported by JavaScript. Date and time values are not one of these fundamental types, but JavaScript does provide a class of object that represents dates and times and can be used to manipulate this type of data. A Date object in JavaScript is created with the `new` operator and the `Date()` constructor (the `new` operator will be introduced in [Chapter 5](#), and we'll learn more about object creation in [Chapter 8](#)):

```
var now = new Date( ); // Create an object holding the current date
and time.
// Create a Date object representing Christmas.
// Note that months are zero-based, so December is month 11!
var xmas = new Date(2000, 11, 25);
```

Methods of the Date object allow you to get and set the various date and time values and to convert the Date to a string, using either local time or GMT time. For example:

```
xmas.setFullYear(xmas.getFullYear( ) + 1); // Change the date to next
Christmas.
var weekday = xmas.getDay( ); // Christmas falls on a Tuesday in
2001.
document.write("Today is: " + now.toLocaleString( )); // Current
date/time.
```

The Date object also defines functions (not methods; they are not invoked through a Date object) to convert a date specified in string or numeric form to an internal millisecond representation that is useful for some kinds of date arithmetic.

You can find full documentation on the Date object and its methods in the core reference section of this book.

3.10 Regular Expressions

Regular expressions provide a rich and powerful syntax for describing textual patterns; they are used for pattern matching and for implementing search and replace operations. JavaScript has adopted the Perl programming language syntax for expressing regular expressions. Regular expression support was first added to the language in JavaScript 1.2 and was standardized and extended by ECMAScript v3.

Regular expressions are represented in JavaScript by the RegExp object and may be created using the `RegExp()` constructor. Like the Date object, the RegExp object is not one of the fundamental data types of JavaScript; it is simply a specialized kind of object provided by all conforming JavaScript implementations.

Unlike the Date object, however, RegExp objects have a literal syntax and can be encoded directly into JavaScript 1.2 programs. Text between a pair of slashes constitutes a regular expression literal. The second slash in the pair can also be followed by one or more letters, which modify the meaning of the pattern. For example:

```
/^HTML/  
/[1-9][0-9]*/  
/\bjavascript\b/i
```

The regular expression grammar is complex and is documented in detail in [Chapter 10](#). At this point, you need only know what a regular expression literal looks like in JavaScript code.

3.11 Error Objects

ECMAScript v3 defines a number of classes that represent errors. The JavaScript interpreter "throws" an object of one of these types when a runtime error occurs. (See the `throw` and `try` statements in [Chapter 6](#) for a discussion of throwing and catching errors.) Each error object has a `message` property that contains an implementation-specific error message. The types of predefined error objects are Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, and URIError. You can find out more about these classes in the core reference section of this book.

3.12 Primitive Data Type Wrapper Objects

When we discussed strings earlier in this chapter, I pointed out a strange feature of that data type: to operate on strings, you use object notation. For example, a typical operation involving strings might look like the following:

```
var s = "These are the times that try people's souls.";
var last_word = s.substring(s.lastIndexOf(" ")+1, s.length);
```

If you didn't know better, it would appear that `s` was an object and that you were invoking methods and reading property values of that object.

What's going on? Are strings objects, or are they primitive data types? The `typeof` operator (see [Chapter 5](#)) assures us that strings have the data type "string", which is distinct from the data type "object". Why, then, are strings manipulated using object notation?

The truth is that a corresponding object class is defined for each of the three key primitive data types. That is, besides supporting the number, string, and boolean data types, JavaScript also supports `Number`, `String`, and `Boolean` classes. These classes are wrappers around the primitive data types. A *wrapper* contains the same primitive data value, but it also defines properties and methods that can be used to manipulate that data.

JavaScript can flexibly convert values from one type to another. When we use a string in an object context -- i.e., when we try to access a property or method of the string -- JavaScript internally creates a `String` wrapper object for the string value. This `String` object is used in place of the primitive string value; the object has properties and methods defined, so the use of the primitive value in an object context succeeds. The same is true, of course, for the other primitive types and their corresponding wrapper objects; we just don't use the other types in an object context nearly as often as we use strings in that context.

Note that the `String` object created when we use a string in an object context is a transient one -- it is used to allow us to access a property or method and then is no longer needed, so it is reclaimed by the system. Suppose `s` is a string and we determine the length of the string with a line like this:

```
var len = s.length;
```

In this case, `s` remains a string; the original string value itself is not changed. A new transient `String` object is created, which allows us to access the `length` property, and then the transient object is discarded, with no change to the original value `s`. If you think this scheme sounds elegant and bizarrely complex at the same time, you are right. Typically, however, JavaScript implementations perform this internal conversion very efficiently, and it is not something you should worry about.

If we want to use a `String` object explicitly in our program, we have to create a nontransient one that is not automatically discarded by the system. `String` objects are created just like other objects, with the `new` operator. For example:

```
var s = "hello world"; // A primitive string value
```

```
var s = new String("Hello World"); // A String object
```

Once we've created a String object `s`, what can we do with it? Nothing that we cannot do with the corresponding primitive string value. If we use the `typeof` operator, it tells us that `s` is indeed an object, and not a string value, but except for that case, we'll find that we can't normally distinguish between a primitive string and the String object.^[6] As we've already seen, strings are automatically converted to String objects whenever necessary. It turns out that the reverse is also true. Whenever we use a String object where a primitive string value is expected, JavaScript automatically converts the String to a string. So if we use our String object with the `+` operator, a transient primitive string value is created so that the string concatenation operation can be performed:

^[6] Note, however, that the `eval()` method treats string values and String objects differently, and it will not behave as you expect it to if you inadvertently pass it a String object instead of a primitive string value.

```
msg = s + '!';
```

Bear in mind that everything we've discussed in this section about string values and String objects applies also to number and boolean values and their corresponding Number and Boolean objects. You can learn more about these classes from their respective entries in the core reference section of this book. In [Chapter 11](#), we'll see more about this primitive type/object duality and about automatic data conversion in JavaScript.

Chapter 4. Variables

A *variable* is a name associated with a value; we say that the variable stores or contains the value. Variables allow you to store and manipulate data in your programs. For example, the following line of JavaScript assigns the value `2` to a variable named `i`:

```
i = 2;
```

And the following line adds `3` to `i` and assigns the result to a new variable, `sum`:

```
var sum = i + 3;
```

These two lines of code demonstrate just about everything you need to know about variables. However, to fully understand how variables work in JavaScript, you need to master a few more concepts. Unfortunately, these concepts require more than a couple of lines of code to explain! The rest of this chapter explains the typing, declaration, scope, contents, and resolution of variables. It also explores garbage collection and the variable/property duality.^[1]

^[1] These are tricky concepts, and a complete understanding of this chapter requires an understanding of concepts introduced in later chapters of the book. If you are relatively new to programming, you may want to read only the first two sections of this chapter and then move on to [Chapter 5](#), [Chapter 6](#), and [Chapter 7](#) before returning to finish up the remainder of this chapter.

4.1 Variable Typing

An important difference between JavaScript and languages such as Java and C is that JavaScript is *untyped*. This means, in part, that a JavaScript variable can hold a value of any data type, unlike a Java or C variable, which can hold only the one particular type of data for which it is declared. For example, it is perfectly legal in JavaScript to assign a number to a variable and then later assign a string to that variable:

```
i = 10;  
i = "ten";
```

In C, C++, Java, or any other strongly typed language, code like this is illegal.

A feature related to JavaScript's lack of typing is that the language conveniently and automatically converts values from one type to another, as necessary. If you attempt to append a number to a string, for example, JavaScript automatically converts the number to the corresponding string so that it can be appended. We'll see more about data type conversion in [Chapter 11](#).

JavaScript is obviously a simpler language for being untyped. The advantage of strongly typed languages such as C++ and Java is that they enforce rigorous programming

practices, which makes it easier to write, maintain, and reuse long, complex programs. Since many JavaScript programs are shorter scripts, this rigor is not necessary and we benefit from the simpler syntax.

4.2 Variable Declaration

Before you use a variable in a JavaScript program, you must *declare* it.^[2] Variables are declared with the `var` keyword, like this:

^[2] If you don't declare a variable explicitly, JavaScript will declare it implicitly for you.

```
var i;  
var sum;
```

You can also declare multiple variables with the same `var` keyword:

```
var i, sum;
```

And you can combine variable declaration with variable initialization:

```
var message = "hello";  
var i = 0, j = 0, k = 0;
```

If you don't specify an initial value for a variable with the `var` statement, the variable is declared, but its initial value is `undefined` until your code stores a value into it.

Note that the `var` statement can also appear as part of the `for` and `for/in` loops (introduced in [Chapter 6](#)), allowing you to succinctly declare the loop variable as part of the loop syntax itself. For example:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");  
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");  
for(var i in o) document.write(i, "<br>");
```

Variables declared with `var` are *permanent*: attempting to delete them with the `delete` operator causes an error. (The `delete` operator is introduced in [Chapter 5](#).)

4.2.1 Repeated and Omitted Declarations

It is legal and harmless to declare a variable more than once with the `var` statement. If the repeated declaration has an initializer, it acts as if it were simply an assignment statement.

If you attempt to read the value of an undeclared variable, JavaScript will generate an error. If you assign a value to a variable that you have not declared with `var`, JavaScript will implicitly declare that variable for you. Note, however, that implicitly declared variables are always created as global variables, even if they are used within the body of a function. To prevent the creation of a global variable (or the use of an existing global variable) when you meant to create a local variable for use within a single function, you must always use the `var` statement within function bodies. It's best to use `var` for all variables, whether global or local. (The distinction between local and global variables is explored in more detail in the next section.)

4.3 Variable Scope

The *scope* of a variable is the region of your program in which it is defined. A *global* variable has global scope -- it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are *local* variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable. For example, the following code prints the word "local":

```
var scope = "global";           // Declare a global variable
function checkscope( ) {
    var scope = "local";       // Declare a local variable with the same
name                             name
    document.write(scope);     // Use the local variable, not the global
one
}
checkscope( );                 // Prints "local"
```

Although you can get away with not using the `var` statement when you write code in the global scope, you must always use `var` to declare local variables. Consider what happens if you don't:

```
scope = "global";              // Declare a global variable, even
without var
function checkscope( ) {
    scope = "local";           // Oops! We just changed the global
variable
    document.write(scope);     // Uses the global variable
    myscope = "local";         // This implicitly declares a new
global variable
    document.write(myscope);   // Uses the new global variable
}
checkscope( );                 // Prints "locallocal"
document.write(scope);         // This prints "local"
document.write(myscope);       // This prints "local"
```

In general, functions do not know what variables are defined in the global scope or what they are being used for. Thus, if a function uses a global variable instead of a local one, it runs the risk of changing a value upon which some other part of the program relies. Fortunately, avoiding this problem is simple: declare all variables with `var`.

In JavaScript 1.2 (and ECMAScript v3), function definitions can be nested. Each function has its own local scope, so it is possible to have several nested layers of local scope. For example:

```
var scope = "global scope";           // A global variable
function checkscope( ) {
    var scope = "local scope";        // A local variable
    function nested( ) {
        var scope = "nested scope";  // A nested scope of local
variables
        document.write(scope);       // Prints "nested scope"
    }
    nested( );
}
checkscope( );
```

4.3.1 No Block Scope

Note that unlike C, C++, and Java, JavaScript does not have block-level scope. All variables declared in a function, no matter where they are declared, are defined *throughout* the function. In the following code, the variables `i`, `j`, and `k` all have the same scope: all three are defined throughout the body of the function. This would not be the case if the code were written in C, C++, or Java:

```
function test(o) {
    var i = 0;                          // i is defined throughout
function
    if (typeof o == "object") {
        var j = 0;                      // j is defined everywhere, not
just block
        for(var k = 0; k < 10; k++) {   // k is defined everywhere, not
just loop
            document.write(k);
        }
        document.write(k);             // k is still defined: prints 10
    }
    document.write(j);                 // j is defined, but may not be
initialized
}
```

The rule that all variables declared in a function are defined throughout the function can cause surprising results. The following code illustrates this:

```

var scope = "global";
function f( ) {
    alert(scope);          // Displays "undefined", not "global"
    var scope = "local";  // Variable initialized here, but defined
everywhere
    alert(scope);          // Displays "local"
}
f( );

```

You might think that the first call to `alert()` would display "global", since the `var` statement declaring the local variable has not yet been executed. Because of the scope rules, however, this is not what happens. The local variable is defined throughout the body of the function, which means the global variable by the same name is hidden throughout the function. Although the local variable is defined throughout, it is not actually initialized until the `var` statement is executed. Thus, the function `f` in the previous example is equivalent to the following:

```

function f( ) {
    var scope;            // Local variable is declared at the start of the
function
    alert(scope);        // It exists here, but still has "undefined"
value
    scope = "local";     // Now we initialize it and give it a value
    alert(scope);        // And here it has a value
}

```

This example illustrates why it is good programming practice to place all of your variable declarations together at the start of any function.

4.3.2 Undefined Versus Unassigned

The examples in the previous section demonstrate a subtle point in JavaScript programming: there are two different kinds of undefined variables. The first kind of undefined variable is one that has never been declared. An attempt to read the value of such an undeclared variable causes a runtime error. *Undeclared variables* are undefined because they simply do not exist. As described earlier, assigning a value to an undeclared variable does not cause an error; instead, it implicitly declares the variable in the global scope.

The second kind of undefined variable is one that has been declared but has never had a value assigned to it. If you read the value of one of these variables, you obtain its default value, `undefined`. This type of undefined variable might more usefully be called *unassigned*, to distinguish it from the more serious kind of undefined variable that has not even been declared and does not exist.

The following code fragment illustrates some of the differences between truly undefined and merely unassigned variables:


```
var x;      // Declare an unassigned variable. Its value is undefined.
alert(u);   // Using an undeclared variable causes an error.
u = 3;      // Assigning a value to an undeclared variable creates the
variable.
```

4.4 Primitive Types and Reference Types

The next topic we need to consider is the content of variables. We often say that variables have or contain values. But just what is it that they contain? To answer this seemingly simple question, we must look again at the data types supported by JavaScript. The types can be divided into two groups: primitive types and reference types. Numbers, boolean values, and the `null` and undefined types are primitive. Objects, arrays, and functions are reference types.

A primitive type has a fixed size in memory. For example, a number occupies eight bytes of memory, and a boolean value can be represented with only one bit. The number type is the largest of the primitive types. If each JavaScript variable reserves eight bytes of memory, the variable can directly hold any primitive value.^[3]

^[3] This is an oversimplification and is not intended as a description of an actual JavaScript implementation.

Reference types are another matter, however. Objects, for example, can be of any length - they do not have a fixed size. The same is true of arrays: an array can have any number of elements. Similarly, a function can contain any amount of JavaScript code. Since these types do not have a fixed size, their values cannot be stored directly in the eight bytes of memory associated with each variable. Instead, the variable stores a *reference* to the value. Typically, this reference is some form of pointer or memory address. It is not the data value itself, but it tells the variable where to look to find the value.

The distinction between primitive and reference types is an important one, as they behave differently. Consider the following code that uses numbers (a primitive type):

```
var a = 3.14; // Declare and initialize a variable
var b = a;    // Copy the variable's value to a new variable
a = 4;       // Modify the value of the original variable
alert(b)     // Displays 3.14; the copy has not changed
```

There is nothing surprising about this code. Now consider what happens if we change the code slightly so that it uses arrays (a reference type) instead of numbers:

```
var a = [1,2,3]; // Initialize a variable to refer to an array
var b = a;      // Copy that reference into a new variable
a[0] = 99;     // Modify the array using the original reference
alert(b);     // Display the changed array [99,2,3] using the new
reference
```

If this result does not seem surprising to you, you're already well familiar with the distinction between primitive and reference types. If it does seem surprising, take a closer look at the second line. Note that it is the reference to the array value, not the array itself, that is being assigned in this statement. After that second line of code, we still have only one array object; we just happen to have two references to it.

If the primitive versus reference type distinction is new to you, just try to keep the variable contents in mind. Variables hold the actual values of primitive types, but they hold only references to the values of reference types. The differing behavior of primitive and reference types is explored in more detail in [Section 11.2](#).

You may have noticed that I did not specify whether strings are primitive or reference types in JavaScript. Strings are an unusual case. They have variable size, so obviously they cannot be stored directly in fixed-size variables. For efficiency, we would expect JavaScript to copy references to strings, not the actual contents of strings. On the other hand, strings behave like a primitive type in many ways. The question of whether strings are a primitive or reference type is actually moot, because strings are *immutable*: there is no way to change the contents of a string value. This means that we cannot construct an example like the previous one that demonstrates that arrays are copied by reference. In the end, it doesn't matter much whether you think of strings as an immutable reference type that behaves like a primitive type or as a primitive type implemented with the internal efficiency of a reference type.

4.5 Garbage Collection

Reference types do not have a fixed size; indeed, some of them can become quite large. As we've already discussed, variables do not directly hold reference values. The value is stored at some other location, and the variables merely hold a reference to that location. Now we need to focus briefly on the actual storage of the value.

Since strings, objects, and arrays do not have a fixed size, storage for them must be allocated dynamically, when the size is known. Every time a JavaScript program creates a string, array, or object, the interpreter must allocate memory to store that entity. Whenever memory is dynamically allocated like this, it must eventually be freed up for reuse, or the JavaScript interpreter will use up all the available memory on the system and crash.

In languages like C and C++, memory must be freed manually. It is the programmer's responsibility to keep track of all the objects that are created and to destroy them (freeing their memory) when they are no longer needed. This can be an onerous task and is often the source of bugs.

Instead of requiring manual deallocation, JavaScript relies on a technique called *garbage collection*. The JavaScript interpreter is able to detect when an object will never again be used by the program. When it determines that an object is unreachable (i.e., there is no longer any way to refer to it using the variables in the program), it knows that the object

is no longer needed and its memory can be reclaimed. Consider the following lines of code, for example:

```
var s = "hello";           // Allocate memory for a string
var u = s.toUpperCase( ); // Create a new string
s = u;                    // Overwrite reference to original string
```

After this code runs, the original string "hello" is no longer reachable -- there are no references to it in any variables in the program. The system detects this fact and frees up its storage space for reuse.

Garbage collection is automatic and is invisible to the programmer. You can create all the garbage objects you want, and the system will clean up after you! You need to know only enough about garbage collection to trust that it works; you don't have to wonder about where all the old objects go. For those who aren't satisfied, however, [Section 11.3](#), contains further details on the JavaScript garbage-collection process.

4.6 Variables as Properties

You may have noticed by now that there are a lot of similarities in JavaScript between variables and the properties of objects. They are both assigned the same way, they are used the same way in JavaScript expressions, and so on. Is there really any fundamental difference between the variable `i` and the property `i` of an object `o`? The answer is no. Variables in JavaScript are fundamentally the same as object properties.

4.6.1 The Global Object

When the JavaScript interpreter starts up, one of the first things it does, before executing any JavaScript code, is create a *global object*. The properties of this object are the global variables of JavaScript programs. When you declare a global JavaScript variable, what you are actually doing is defining a property of the global object.

The JavaScript interpreter initializes the global object with a number of properties that refer to predefined values and functions. For example, the `Infinity`, `parseInt`, and `Math` properties refer to the number infinity, the predefined `parseInt()` function, and the predefined `Math` object. You can read about these global values in the core reference section of this book.

In top-level code (i.e., JavaScript code that is not part of a function), you can use the JavaScript keyword `this` to refer to the global object. Within functions, `this` has a different use, which is described in [Chapter 7](#).

In client-side JavaScript, the `Window` object serves as the global object for all JavaScript code contained in the browser window it represents. This global `Window` object has a self-referential `window` property that can be used instead of `this` to refer to the global

object. The Window object defines the core global properties, such as `parseInt` and `Math`, and also global client-side properties, such as `navigator` and `screen`.

4.6.2 Local Variables: The Call Object

If global variables are properties of the special global object, then what are local variables? They too are properties of an object. This object is known as the *call object*. The call object has a shorter life span than the global object, but it serves the same purpose. While the body of a function is executing, the function arguments and local variables are stored as properties of this call object. The use of an entirely separate object for local variables is what allows JavaScript to keep local variables from overwriting the value of global variables with the same name.

4.6.3 JavaScript Execution Contexts

Each time the JavaScript interpreter begins to execute a function, it creates a new *execution context* for that function. An execution context is, obviously, the context in which any piece of JavaScript code executes. An important part of the context is the object in which variables are defined. Thus, JavaScript code that is not part of any function runs in an execution context that uses the global object for variable definitions. And every JavaScript function runs in its own unique execution context with its own call object in which local variables are defined.

An interesting point to note is that JavaScript implementations may allow multiple global execution contexts, each with a *different* global object. (Although, in this case, each global object is not entirely global.)^[4] The obvious example is client-side JavaScript, in which each separate browser window, or each frame within a window, defines a separate global execution context. Client-side JavaScript code in each frame or window runs in its own execution context and has its own global object. However, these separate client-side global objects have properties that link them to one another. Thus, JavaScript code in one frame might refer to another frame with the expression `parent.frames[1]`, and the global variable `x` in the first frame might be referenced by the expression `parent.frames[0].x` in the second frame.

^[4] This is merely an aside; if it does not interest you, feel free to move on to the next section.

You don't need to fully understand how separate window and frame execution contexts are linked together in client-side JavaScript right now. We'll cover that topic in detail when we discuss the integration of JavaScript with web browsers in [Chapter 12](#). What you should understand now is that JavaScript is flexible enough that a single JavaScript interpreter can run scripts in different global execution contexts and that those contexts need not be entirely separate -- they can refer back and forth to each other.

This last point requires additional consideration. When JavaScript code in one execution context can read and write property values and execute functions that are defined in another execution context, we've reached a level of complexity that requires

consideration of security issues. Take client-side JavaScript as an example. Suppose browser window A is running a script or contains information from your local intranet, and window B is running a script from some random site out on the Internet. In general, we do not want to allow the code in window B to be able to access the properties of window A. If we allow it to do this, it might be able to read sensitive company information and steal it, for example. Thus, in order to safely run JavaScript code, there must be a security mechanism that prevents access from one execution context to another when such access should not be permitted. We'll return to this topic in [Chapter 21](#).

4.7 Variable Scope Revisited

When we first discussed the notion of variable scope, I based the definition solely on the lexical structure of JavaScript code: global variables have global scope and variables declared in functions have local scope. If one function definition is nested within another, variables declared within that nested function have a nested local scope. Now that we know that global variables are properties of a global object and that local variables are properties of a special call object, we can return to the notion of variable scope and reconceptualize it. This new description of scope offers a useful way to think about variables in many contexts; it provides a powerful new understanding of how JavaScript works.

Every JavaScript execution context has a *scope chain* associated with it. This scope chain is a list or chain of objects. When JavaScript code needs to look up the value of a variable x (a process called *variable name resolution*), it starts by looking at the first object in the chain. If that object has a property named x , the value of that property is used. If the first object does not have a property named x , JavaScript continues the search with the next object in the chain. If the second object does not have a property named x , the search moves on to the next object, and so on.

In top-level JavaScript code (i.e., code not contained within any function definitions), the scope chain consists of a single object, the global object. All variables are looked up in this object. If a variable does not exist, the variable value is undefined. In a (non-nested) function, however, the scope chain consists of two objects. The first is the function's call object, and the second is the global object. When the function refers to a variable, the call object (the local scope) is checked first, and the global object (the global scope) is checked second. A nested function would have three or more objects in its scope chain. [Figure 4-1](#) illustrates the process of looking up a variable name in the scope chain of a function.

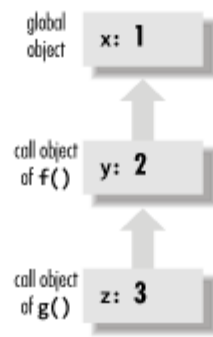
Figure 4-1. The scope chain and variable resolution

Lexical Scope

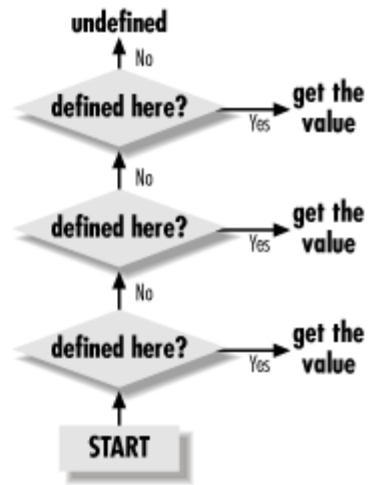
```
var x=1;

function f() {
  var y=2;
  function g() {
    var z=3;
  }
}
```

Scope Chain



Variable Lookup



Chapter 5. Expressions and Operators

This chapter explains how expressions and operators work in JavaScript. If you are familiar with C, C++, or Java, you'll notice that the expressions and operators in JavaScript are very similar, and you'll be able to skim this chapter quickly. If you are not a C, C++, or Java programmer, this chapter tells you everything you need to know about expressions and operators in JavaScript.

5.1 Expressions

An *expression* is a phrase of JavaScript that a JavaScript interpreter can *evaluate* to produce a value. The simplest expressions are literals or variable names, like these:

```
1.7           // A numeric literal
"JavaScript is fun!" // A string literal
true          // A boolean literal
null         // The literal null value
/java/       // A regular expression literal
{ x:2, y:2 }  // An object literal
[2,3,5,7,11,13,17,19] // An array literal
function(x){return x*x;} // A function literal
i            // The variable i
sum         // The variable sum
```

The value of a literal expression is simply the literal value itself. The value of a variable expression is the value that the variable contains or refers to.

These expressions are not particularly interesting. More complex (and interesting) expressions can be created by combining simple expressions. For example, we saw that `1.7` is an expression and `i` is an expression. The following is also an expression:

```
i + 1.7
```

The value of this expression is determined by adding the values of the two simpler expressions. The `+` in this example is an *operator* that is used to combine two expressions into a more complex expression. Another operator is `-`, which is used to combine expressions by subtraction. For example:

```
(i + 1.7) - sum
```

This expression uses the `-` operator to subtract the value of the `sum` variable from the value of our previous expression, `i + 1.7`. JavaScript supports a number of other operators besides `+` and `-`, as you'll see in the next section.

5.2 Operator Overview

If you are a C, C++, or Java programmer, most of the JavaScript operators should already be familiar to you. [Table 5-1](#) summarizes the operators; you can refer to this table for reference. Note that most operators are represented by punctuation characters such as `+` and `=`. Some, however, are represented by keywords such as `delete` and `instanceof`. Keyword operators are regular operators, just like those expressed with punctuation; they are simply expressed using a more readable and less succinct syntax.

In this table, the column labeled "P" gives the operator precedence and the column labeled "A" gives the operator associativity, which can be L (left-to-right) or R (right-to-left). If you do not already understand precedence and associativity, the subsections that follow the table explain these concepts. The operators themselves are documented following that discussion.

P	A	Operator	Operand type(s)	Operation performed
15	L	<code>.</code>	object, identifier	Property access
	L	<code>[]</code>	array, integer	Array index
	L	<code>()</code>	function, arguments	Function call
	R	<code>new</code>	constructor call	Create new object
14	R	<code>++</code>	lvalue	Pre- or post-increment (unary)
	R	<code>--</code>	lvalue	Pre- or post-decrement (unary)
	R	<code>-</code>	number	Unary minus (negation)
	R	<code>+</code>	number	Unary plus (no-op)
	R	<code>~</code>	integer	Bitwise complement (unary)
	R	<code>!</code>	boolean	Logical complement (unary)

Table 5-1. JavaScript operators

P	A	Operator	Operand type(s)	Operation performed
	R	<code>delete</code>	lvalue	Undefine a property (unary)
	R	<code>typeof</code>	any	Return data type (unary)
	R	<code>void</code>	any	Return undefined value (unary)
13	L	<code>*, /, %</code>	numbers	Multiplication, division, remainder
12	L	<code>+, -</code>	numbers	Addition, subtraction
	L	<code>+</code>	strings	String concatenation
11	L	<code><<</code>	integers	Left shift
	L	<code>>></code>	integers	Right shift with sign-extension
	L	<code>>>></code>	integers	Right shift with zero extension
10	L	<code><, <=</code>	numbers or strings	Less than, less than or equal
	L	<code>>, >=</code>	numbers or strings	Greater than, greater than or equal
	L	<code>instanceof</code>	object, constructor	Check object type
	L	<code>in</code>	string, object	Check whether property exists
9	L	<code>==</code>	any	Test for equality
	L	<code>!=</code>	any	Test for inequality
	L	<code>===</code>	any	Test for identity
	L	<code>!==</code>	any	Test for non-identity
8	L	<code>&</code>	integers	Bitwise AND
7	L	<code>^</code>	integers	Bitwise XOR
6	L	<code> </code>	integers	Bitwise OR
5	L	<code>&&</code>	booleans	Logical AND

Table 5-1. JavaScript operators

P	A	Operator	Operand type(s)	Operation performed
4	L		booleans	Logical OR
3	R	?:	boolean, any, any	Conditional operator (3 operands)
2	R	=	lvalue, any	Assignment
	R	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =	lvalue, any	Assignment with operation
1	L	,	any	Multiple evaluation

5.2.1 Number of Operands

Operators can be categorized based on the number of operands they expect. Most JavaScript operators, like the + operator we saw earlier, are *binary operators* that combine two expressions into a single, more complex expression. That is, they operate on two operands. JavaScript also supports a number of *unary operators*, which convert a single expression into a single, more complex expression. The - operator in the expression -3 is a unary operator that performs the operation of negation on the operand 3. Finally, JavaScript supports one *ternary operator*, the conditional operator ?:, which combines the value of three expressions into a single expression.

5.2.2 Type of Operands

When constructing JavaScript expressions, you must pay attention to the data types that are being passed to operators and to the data types that are returned. Different operators expect their operands' expressions to evaluate to values of a certain data type. For example, it is not possible to multiply strings, so the expression "a" * "b" is not legal in JavaScript. Note, however, that JavaScript tries to convert expressions to the appropriate type whenever possible, so the expression "3" * "5" is legal. Its value is the number 15, not the string "15". We'll consider JavaScript type conversions in detail in [Section 11.1](#).

Furthermore, some operators behave differently depending on the type of the operands. Most notably, the + operator adds numeric operands but concatenates string operands. Also, if passed one string and one number, it converts the number to a string and concatenates the two resulting strings. For example, "1" + 0 yields the string "10".

Notice that the assignment operators, as well as a few other operators, expect their lefthand expressions to be lvalues. *lvalue* is a historical term that means "an expression that can legally appear on the lefthand side of an assignment expression." In JavaScript,

variables, properties of objects, and elements of arrays are lvalues. The ECMAScript specification allows built-in functions to return lvalues but does not define any built-in functions that behave that way.

Finally, note that operators do not always return the same type as their operands. The comparison operators (less than, equal to, greater than, etc.) take operands of various types, but when comparison expressions are evaluated, they always return a boolean result that indicates whether the comparison is true or not. For example, the expression `a < 3` returns `true` if the value of variable `a` is in fact less than `3`. As we'll see, the boolean values returned by comparison operators are used in `if` statements, `while` loops, and `for` loops -- JavaScript statements that control the execution of a program based on the results of evaluating expressions that contain comparison operators.

5.2.3 Operator Precedence

In [Table 5-1](#), the column labeled "P" specifies the *precedence* of each operator. Operator precedence controls the order in which operations are performed. Operators with higher numbers in the "P" column are performed before those with lower numbers.

Consider the following expression:

```
w = x + y*z;
```

The multiplication operator `*` has a higher precedence than the addition operator `+`, so the multiplication is performed before the addition. Furthermore, the assignment operator `=` has the lowest precedence, so the assignment is performed after all the operations on the righthand side are completed.

Operator precedence can be overridden with the explicit use of parentheses. To force the addition in the previous example to be performed first, we would write:

```
w = (x + y)*z;
```

In practice, if you are at all unsure about the precedence of your operators, the simplest thing is to use parentheses to make the evaluation order explicit. The only rules that are important to know are these: multiplication and division are performed before addition and subtraction, and assignment has very low precedence and is almost always performed last.

5.2.4 Operator Associativity

In [Table 5-1](#), the column labeled "A" specifies the *associativity* of the operator. A value of L specifies left-to-right associativity, and a value of R specifies right-to-left associativity. The associativity of an operator specifies the order in which operations of

the same precedence are performed. Left-to-right associativity means that operations are performed from left to right. For example, the addition operator has left-to-right associativity, so:

```
w = x + y + z;
```

is the same as:

```
w = ((x + y) + z);
```

On the other hand, the following (almost nonsensical) expressions:

```
x = ~~~y;  
w = x = y = z;  
q = a?b:c?d:e?f:g;
```

are equivalent to:

```
x = ~(-(~y));  
w = (x = (y = z));  
q = a?b:(c?d:(e?f:g));
```

because the unary, assignment, and ternary conditional operators have right-to-left associativity.

5.3 Arithmetic Operators

Having explained operator precedence, associativity, and other background material, we can start to discuss the operators themselves. This section details the arithmetic operators:

Addition (+)

The `+` operator adds numeric operands or concatenates string operands. If one operand is a string, the other is converted to a string and the two strings are then concatenated. Object operands are converted to numbers or strings that can be added or concatenated. The conversion is performed by the `valueOf()` method and/or the `toString()` method of the object.

Subtraction (-)

When `-` is used as a binary operator, it subtracts its second operand from its first operand. If used with non-numeric operands, it attempts to convert them to numbers.

Multiplication ()*

The `*` operator multiplies its two operands. If used with non-numeric operands, it attempts to convert them to numbers.

Division (/)

The `/` operator divides its first operand by its second. If used with non-numeric operands, it attempts to convert them to numbers. If you are used to programming languages that distinguish between integer and floating-point numbers, you might expect to get an integer result when you divide one integer by another. In JavaScript, however, all numbers are floating-point, so all divisions have floating-point results: `5/2` evaluates to `2.5`, not `2`. Division by zero yields positive or negative infinity, while `0/0` evaluates to `NaN`.

Modulo (%)

The `%` operator computes the first operand modulo the second operand. That is, it returns the remainder when the first operand is divided by the second operand an integral number of times. If used with non-numeric operands, the modulo operator attempts to convert them to numbers. The sign of the result is the same as the sign of the first operand. For example, `5 % 2` evaluates to `1`.

While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, `-4.3 % 2.1` evaluates to `-0.1`.

Unary minus (-)

When `-` is used as a unary operator, before a single operand, it performs unary negation. In other words, it converts a positive value to an equivalently negative value, and vice versa. If the operand is not a number, this operator attempts to convert it to one.

Unary plus (+)

For symmetry with the unary minus operator, JavaScript also has a unary plus operator. This operator allows you to explicitly specify the sign of numeric literals, if you feel that this will make your code clearer:

```
var profit = +1000000;
```

In code like this, the `+` operator does nothing; it simply evaluates to the value of its argument. Note, however, that for non-numeric arguments, the `+` operator has the effect of converting the argument to a number. It returns `NaN` if the argument cannot be converted.

Increment (++)

The `++` operator increments (i.e., adds 1 to) its single operand, which must be a variable, an element of an array, or a property of an object. If the value of this variable, element, or property is not a number, the operator first attempts to convert it to one. The precise behavior of this operator depends on its position relative to the operand. When used before the operand, where it is known as the pre-increment operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the post-increment operator, it increments its operand but evaluates to the *unincremented* value of that operand. If the value to be incremented is not a number, it is converted to one by this process.

For example, the following code sets both `i` and `j` to 2:

```
i = 1;  
j = ++i;
```

But these lines set `i` to 2 and `j` to 1:

```
i = 1;  
j = i++;
```

This operator, in both of its forms, is most commonly used to increment a counter that controls a loop. Note that, because of JavaScript's automatic semicolon insertion, you may not insert a line break between the post-increment or post-decrement operator and the operand that precedes it. If you do so, JavaScript will treat the operand as a complete statement by itself and will insert a semicolon before it.

Decrement (--)

The `--` operator decrements (i.e., subtracts 1 from) its single numeric operand, which must be a variable, an element of an array, or a property of an object. If the value of this variable, element, or property is not a number, the operator first attempts to convert it to one. Like the `++` operator, the precise behavior of `--` depends on its position relative to the operand. When used before the operand, it decrements and returns the decremented value. When used after the operand, it decrements the operand but returns the *undecremented* value.

5.4 Equality Operators

This section describes the JavaScript equality and inequality operators. These are operators that compare two values to determine whether they are the same or different and return a boolean value (`true` or `false`) depending on the result of the comparison. As we'll see in [Chapter 6](#), they are most commonly used in things like `if` statements and `for` loops, to control the flow of program execution.

5.4.1 Equality (`==`) and Identity (`===`)

The `==` and `===` operators check whether two values are the same, using two different definitions of sameness. Both operators accept operands of any type, and both return `true` if their operands are the same and `false` if they are different. The `===` operator is known as the identity operator, and it checks whether its two operands are "identical" using a strict definition of sameness. The `==` operator is known as the equality operator; it checks whether its two operands are "equal" using a more relaxed definition of sameness that allows type conversions.

The identity operator is standardized by ECMAScript v3 and implemented in JavaScript 1.3 and later. With the introduction of the identity operator, JavaScript supports `=`, `==`, and `===` operators. Be sure you understand the differences between the assignment, equality, and identity operators, and be careful to use the right one when coding! Although it is tempting to call all three operators "equals," it may help to reduce confusion if you read "gets or is assigned" for `=`, "is equal to" for `==`, and "is identical to" for `===`.

In JavaScript, numbers, strings, and boolean values are compared *by value*. In this case, two separate values are involved, and the `==` and `===` operators check that these two values are identical. This means that two variables are equal or identical only if they contain the same value. For example, two strings are equal only if they each contain exactly the same characters.

On the other hand, objects, arrays, and functions are compared *by reference*. This means that two variables are equal only if they *refer to* the same object. Two separate arrays are never equal or identical, even if they contain equal or identical elements. Two variables that contain references to objects, arrays, or functions are equal only if they refer to the same object, array, or function. If you want to test that two distinct objects contain the

same properties or that two distinct arrays contain the same elements, you'll have to check the properties or elements individually for equality or identity. (And, if any of the properties or elements are themselves objects or arrays, you'll have to decide how deep you want the comparison to go.)

The following rules are used to determine whether two values are identical according to the `===` operator:

- If the two values have different types, they are not identical.
- If both values are numbers and have the same value, they are identical, unless either or both values are `NaN`, in which case they are not identical. The `NaN` value is never identical to any other value, including itself! To check whether a value is `NaN`, use the global `isNaN()` function.
- If both values are strings and contain exactly the same characters in the same positions, they are identical. If the strings differ in length or content, they are not identical. Note that in some cases, the Unicode standard allows more than one way to encode the same string. For efficiency, however, JavaScript string comparison compares strictly on a character-by-character basis, and it assumes that all strings have been converted to a "normalized form" before they are compared. See the "`String.localeCompare()`" reference page in the core reference section of this book for another way to compare strings.
- If both values are the boolean value `true` or both are the boolean value `false`, they are identical.
- If both values refer to the same object, array, or function, they are identical. If they refer to different objects (or arrays or functions) they are not identical, even if both objects have identical properties or both arrays have identical elements.
- If both values are `null` or both values are `undefined`, they are identical.

The following rules are used to determine whether two values are equal according to the `==` operator:

- If the two values have the same type, test them for identity. If the values are identical, they are equal; if they are not identical, they are not equal.
- If the two values do not have the same type, they may still be equal. Use the following rules and type conversions to check for equality:
 - If one value is `null` and the other is `undefined`, they are equal.
 - If one value is a number and the other is a string, convert the string to a number and try the comparison again, using the converted value.
 - If either value is `true`, convert it to 1 and try the comparison again. If either value is `false`, convert it to 0 and try the comparison again.
 - If one value is an object and the other is a number or string, convert the object to a primitive and try the comparison again. An object is converted to a primitive value by either its `toString()` method or its `valueOf()` method. The built-in classes of core JavaScript attempt `valueOf()` conversion before `toString()` conversion, except for the `Date` class, which performs `toString()` conversion. Objects that are not part of core

JavaScript may convert themselves to primitive values in an implementation-defined way.

- Any other combinations of values are not equal.

As an example of testing for equality, consider the comparison:

```
"1" == true
```

This expression evaluates to `true`, indicating that these very different-looking values are in fact equal. The boolean value `true` is first converted to the number 1, and the comparison is done again. Next, the string "1" is converted to the number 1. Since both numbers are now the same, the comparison returns `true`.

When the equality operator in JavaScript 1.1 attempted to convert a string to a number and failed, it displayed an error message noting that the string could not be converted, instead of converting the string to `NaN` and returning `false` as the result of the comparison. This bug has been fixed in JavaScript 1.2.

5.4.1.1 Equality and inequality in Netscape

The `==` operator always behaves as described previously, and the `!=` operator always behaves as described in the next section, with one exception. In client-side JavaScript in Netscape 4 and later, when embedded in a `<script>` tag that explicitly specifies JavaScript 1.2 as its `language` attribute, the equality operator behaves like the identity operator, and the inequality operator behaves like the non-identity operator. To avoid this incompatibility, never use the `language="JavaScript1.2"` attribute to embed your client-side JavaScript code. See [Section 11.6](#), for a complete list of similar JavaScript 1.2 incompatibilities.

5.4.2 Inequality (!=) and Non-Identity (!==)

The `!=` and `!==` operators test for the exact opposite of the `==` and `===` operators. The `!=` inequality operator returns `false` if two values are equal to each other and returns `true` otherwise. The `!==` non-identity operator returns `false` if two values are identical to each other and returns `true` otherwise. Note that this operator is standardized by ECMAScript v3 and implemented in JavaScript 1.3 and later.

As we'll see, the `!` operator computes the Boolean NOT operation. This makes it easy to remember that `!=` stands for "not equal to" and `!==` stands for "not identical to." See the previous section for details on how equality and identity are defined for different data types.

5.5 Relational Operators

This section describes the JavaScript relational operators. These are operators that test for a relationship (such as "less-than" or "property-of") between two values and return `true` or `false` depending on whether that relationship exists. As we'll see in [Chapter 6](#), they are most commonly used in things like `if` statements and `while` loops, to control the flow of program execution.

5.5.1 Comparison Operators

The most commonly used types of relational operators are the comparison operators, which are used to determine the relative order of two values. The comparison operators are:

Less than (<)

The `<` operator evaluates to `true` if its first operand is less than its second operand; otherwise it evaluates to `false`.

Greater than (>)

The `>` operator evaluates to `true` if its first operand is greater than its second operand; otherwise it evaluates to `false`.

Less than or equal (<=)

The `<=` operator evaluates to `true` if its first operand is less than or equal to its second operand; otherwise it evaluates to `false`.

Greater than or equal (>=)

The `>=` operator evaluates to `true` if its first operand is greater than or equal to its second operand; otherwise it evaluates to `false`.

The operands of these comparison operators may be of any type. Comparison can be performed only on numbers and strings, however, so operands that are not numbers or strings are converted. Comparison and conversion occur as follows:

- If both operands are numbers, or if both convert to numbers, they are compared numerically.
- If both operands are strings or convert to strings, they are compared as strings.
- If one operand is or converts to a string and one is or converts to a number, the operator attempts to convert the string to a number and perform a numerical comparison. If the string does not represent a number, it converts to `NaN`, and the

comparison is `false`. (In JavaScript 1.1, the string-to-number conversion causes an error instead of yielding `NaN`.)

- If an object can be converted to either a number or a string, JavaScript performs the numerical conversion. This means, for example, that `Date` objects are compared numerically, and it is meaningful to compare two dates to see whether one is earlier than the other.
- If the operands of the comparison operators cannot both be successfully converted to numbers or to strings, these operators always return `false`.
- If either operand is or converts to `NaN`, the comparison operator always yields `false`.

Keep in mind that string comparison is done on a strict character-by-character basis, using the numerical value of each character from the Unicode encoding. Although in some cases the Unicode standard allows equivalent strings to be encoded using different sequences of characters, the JavaScript comparison operators do not detect these encoding differences; they assume that all strings are expressed in normalized form. Note in particular that string comparison is case-sensitive, and in the Unicode encoding (at least for the ASCII subset), all capital letters are "less than" all lowercase letters. This rule can cause confusing results if you do not expect it. For example, according to the `<` operator, the string `"Zoo"` is less than the string `"aardvark"`.

For a more robust string comparison algorithm, see the `String.localeCompare()` method, which also takes locale-specific definitions of "alphabetical order" into account. For case-insensitive comparisons, you must first convert the strings to all lowercase or all uppercase using `String.toLowerCase()` or `String.toUpperCase()`.

The `<=` (less-than-or-equal) and `>=` (greater-than-or-equal) operators do not rely on the equality or identity operators for determining whether two values are "equal." Instead, the less-than-or-equal operator is simply defined as "not greater than," and the greater-than-or-equal operator is defined as "not less than." The one exception is when either operand is (or converts to) `NaN`, in which case all four comparison operators return `false`.

5.5.2 The `in` Operator

The `in` operator expects a lefthand operand that is or can be converted to a string. It expects a righthand operand that is an object (or array). It evaluates to `true` if the lefthand value is the name of a property of the righthand object. For example:

```
var point = { x:1, y:1 };           // Define an object
var has_x_coord = "x" in point;    // Evaluates to true
var has_y_coord = "y" in point;    // Evaluates to true
var has_z_coord = "z" in point;    // Evaluates to false; not a 3-D point
var ts = "toString" in point;      // Inherited property; evaluates to
true
```

5.5.3 The instanceof Operator

The `instanceof` operator expects a lefthand operand that is an object and a righthand operand that is the name of a class of objects. The operator evaluates to `true` if the lefthand object is an instance of the righthand class and evaluates to `false` otherwise. We'll see in [Chapter 8](#) that, in JavaScript, classes of objects are defined by the constructor function that is used to initialize them. Thus, the righthand operand of `instanceof` should be the name of a constructor function. Note that all objects are instances of `Object`. For example:

```
var d = new Date(); // Create a new object with the Date( )
                    constructor
d instanceof Date; // Evaluates to true; d was created with Date( )
d instanceof Object; // Evaluates to true; all objects are instances
                    of Object
d instanceof Number; // Evaluates to false; d is not a Number object
var a = [1, 2, 3]; // Create an array with array literal syntax
a instanceof Array; // Evaluates to true; a is an array
a instanceof Object; // Evaluates to true; all arrays are objects
a instanceof RegExp; // Evaluates to false; arrays are not regular
                    expressions
```

If the lefthand operand of `instanceof` is not an object, or if the righthand operand is an object that is not a constructor function, `instanceof` returns false. On the other hand, it returns a runtime error if the righthand operand is not an object at all.

5.6 String Operators

As we've discussed in the previous sections, there are several operators that have special effects when their operands are strings.

The `+` operator concatenates two string operands. That is, it creates a new string that consists of the first string followed by the second. For example, the following expression evaluates to the string "hello there":

```
"hello" + " " + "there"
```

And the following lines produce the string "22":

```
a = "2"; b = "2";
c = a + b;
```

The `<`, `<=`, `>`, and `>=` operators compare two strings to determine what order they fall in. The comparison uses alphabetical order. As noted above, however, this alphabetical order is based on the Unicode character encoding used by JavaScript. In this encoding, all

capital letters in the Latin alphabet come before (are less than) all lowercase letters, which can cause unexpected results.

The `==` and `!=` operators work on strings, but, as we've seen, these operators work for all data types, and they do not have any special behavior when used with strings.

The `+` operator is a special one -- it gives priority to string operands over numeric operands. As noted earlier, if either operand to `+` is a string (or an object), the other operand is converted to a string (or both operands are converted to strings) and concatenated, rather than added. On the other hand, the comparison operators perform string comparison only if *both* operands are strings. If only one operand is a string, JavaScript attempts to convert it to a number. The following lines illustrate:

```
1 + 2           // Addition. Result is 3.
"1" + "2"      // Concatenation. Result is "12".
"1" + 2        // Concatenation; 2 is converted to "2". Result is "12".
11 < 3         // Numeric comparison. Result is false.
"11" < "3"     // String comparison. Result is true.
"11" < 3       // Numeric comparison; "11" converted to 11. Result is
false.
"one" < 3      // Numeric comparison; "one" converted to NaN. Result is
false.
               // In JavaScript 1.1, this causes an error instead of NaN.
```

Finally, it is important to note that when the `+` operator is used with strings and numbers, it may not be associative. That is, the result may depend on the order in which operations are performed. This can be seen with examples like these:

```
s = 1 + 2 + " blind mice"; // Yields "3 blind mice"
t = "blind mice: " + 1 + 2; // Yields "blind mice: 12"
```

The reason for this surprising difference in behavior is that the `+` operator works from left to right, unless parentheses change this order. Thus, the last two examples are equivalent to these:

```
s = (1 + 2) + "blind mice"; // 1st + yields number; 2nd yields
string
t = ("blind mice: " + 1) + 2; // Both operations yield strings
```

5.7 Logical Operators

The logical operators are typically used to perform Boolean algebra. They are often used in conjunction with comparison operators to express complex comparisons that involve more than one variable and are frequently used with the `if`, `while`, and `for` statements.

5.7.1 Logical AND (&&)

When used with boolean operands, the `&&` operator performs the Boolean AND operation on the two values: it returns `true` if and only if both its first operand *and* its second operand are `true`. If one or both of these operands is `false`, it returns `false`.

The actual behavior of this operator is somewhat more complicated. It starts by evaluating its first operand, the expression on its left. If the value of this expression can be converted to `false` (for example, if the left operand evaluates to `null`, `0`, `"`, or `undefined`), the operator returns the value of the lefthand expression. Otherwise, it evaluates its second operand, the expression on its right, and returns the value of that expression.^[1]

^[1] In JavaScript 1.0 and JavaScript 1.1, if the lefthand expression evaluates to `false`, the `&&` operator returns `false`, rather than returning the unconverted value of the lefthand expression.

Note that, depending on the value of the lefthand expression, this operator may or may not evaluate the righthand expression. You may occasionally see code that purposely exploits this feature of the `&&` operator. For example, the following two lines of JavaScript code have equivalent effects:

```
if (a == b) stop( );  
(a == b) && stop( );
```

While some programmers (particularly Perl programmers) find this a natural and useful programming idiom, I recommend against using it. The fact that the righthand side is not guaranteed to be evaluated is a frequent source of bugs. Consider the following code, for example:

```
if ((a == null) && (b++ > 10)) stop( );
```

This statement probably does not do what the programmer intended, since the increment operator on the righthand side is not evaluated whenever the comparison on the lefthand side is `false`. To avoid this problem, do not use expressions with side effects (assignments, increments, decrements, and function calls) on the righthand side of `&&` unless you are quite sure you know exactly what you are doing.

Despite the fairly confusing way that this operator actually works, it is easiest, and perfectly safe, to think of it as merely a Boolean algebra operator. Although it does not actually return a boolean value, the value it returns can always be converted to a boolean value.

5.7.2 Logical OR (||)

When used with boolean operands, the `||` operator performs the Boolean OR operation on the two values: it returns `true` if either the first operand *or* the second operand is `true`, or if both are `true`. If both operands are `false`, it returns `false`.

Although the `||` operator is most often used simply as a Boolean OR operator, it, like the `&&` operator, has more complex behavior. It starts by evaluating its first operand, the expression on its left. If the value of this expression can be converted to `true`, it returns the value of the lefthand expression. Otherwise, it evaluates its second operand, the expression on its right, and returns the value of that expression.^[2]

^[2] In JavaScript 1.0 and JavaScript 1.1, if the lefthand expression could be converted to `true`, the operator returns `true` rather than returning the unconverted value of the lefthand expression.

As with the `&&` operator, you should avoid righthand operands that include side effects, unless you purposely want to make use of the fact that the righthand expression may not be evaluated.

Even when the `||` operator is used with operands that are not boolean values, it can still be considered a Boolean OR operator, since its return value, whatever the type, can be converted to a boolean value.

5.7.3 Logical NOT (!)

The `!` operator is a unary operator; it is placed before a single operand. Its purpose is to invert the boolean value of its operand. For example, if the variable `a` has the value `true` (or is a value that converts to `true`), `!a` has the value `false`. And if the expression `p && q` evaluates to `false` (or to a value that converts to `false`), `!(p && q)` evaluates to `true`. Note that you can convert any value `x` to a boolean value by applying this operator twice: `!!x`.

5.8 Bitwise Operators

Despite the fact that all numbers in JavaScript are floating-point, the bitwise operators require numeric operands that have integer values. They operate on these integer operands using a 32-bit integer representation instead of the equivalent floating-point representation. Four of these operators perform Boolean algebra on the individual bits of the operands, behaving as if each bit in each operand were a boolean value and performing similar operations to those performed by the logical operators we saw earlier. The other three bitwise operators are used to shift bits left and right.

In JavaScript 1.0 and JavaScript 1.1, the bitwise operators return `NaN` if used with operands that are not integers or that are too large to fit in a 32-bit integer representation. JavaScript 1.2 and ECMAScript, however, simply coerce the operands to 32-bit integers by dropping any fractional part of the operand or any bits beyond the 32nd. The shift

operators require a righthand operand between 0 and 31. After converting this operand to a 32-bit integer as described earlier, they drop any bits beyond the 5th, which yields a number in the appropriate range.

If you are not familiar with binary numbers and the binary representation of decimal integers, you can skip the operators described in this section. The purpose of these operators is not described here; they are needed for low-level manipulation of binary numbers and are not commonly used in JavaScript programming. The bitwise operators are:

Bitwise AND (&)

The `&` operator performs a Boolean AND operation on each bit of its integer arguments. A bit is set in the result only if the corresponding bit is set in both operands. For example, `0x1234 & 0x00FF` evaluates to `0x0034`.

Bitwise OR (|)

The `|` operator performs a Boolean OR operation on each bit of its integer arguments. A bit is set in the result if the corresponding bit is set in one or both of the operands. For example, `9 | 10` evaluates to `11`.

Bitwise XOR (^)

The `^` operator performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. A bit is set in this operation's result if a corresponding bit is set in one (but not both) of the two operands. For example, `9 ^ 10` evaluates to `3`.

Bitwise NOT (~)

The `~` operator is a unary operator that appears before its single integer argument. It operates by reversing all bits in the operand. Because of the way signed integers are represented in JavaScript, applying the `~` operator to a value is equivalent to changing its sign and subtracting 1. For example `~0x0f` evaluates to `0xffffffff0`, or `-16`.

Shift left (<<)

The `<<` operator moves all bits in its first operand to the left by the number of places specified in the second operand, which should be an integer between 0 and 31. For example, in the operation `a << 1`, the first bit (the ones bit) of `a` becomes the second bit (the twos bit), the second bit of `a` becomes the third, etc. A zero is used for the new first bit, and the value of the 32nd bit is lost. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, etc. For example, `7 << 1` evaluates to 14.

Shift right with sign (>>)

The `>>` operator moves all bits in its first operand to the right by the number of places specified in the second operand (an integer between 0 and 31). Bits that are shifted off the right are lost. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits. Shifting a value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on. For example, `7 >> 1` evaluates to 3 and `-7 >> 1` evaluates to -4.

Shift right with zero fill (>>>)

The `>>>` operator is just like the `>>` operator, except that the bits shifted in on the left are always zero, regardless of the sign of the first operand. For example, `-1 >> 4` evaluates to -1, but `-1 >>> 4` evaluates to 268435455 (0x0fffffff).

5.9 Assignment Operators

As we saw in the discussion of variables in [Chapter 4](#), `=` is used in JavaScript to assign a value to a variable. For example:

```
i = 0
```

While you might not normally think of such a line of JavaScript as an expression that can be evaluated, it is in fact an expression and, technically speaking, `=` is an operator.

The `=` operator expects its lefthand operand to be either a variable, the element of an array, or a property of an object. It expects its righthand operand to be an arbitrary value of any type. The value of an assignment expression is the value of the righthand operand. As a side effect, the `=` operator assigns the value on the right to the variable, element, or property on the left, so that future uses of the variable, element, or property refer to the value.

Because `=` is defined as an operator, you can include it in more complex expressions. For example, you can assign and test a value in the same expression with code like this:

```
(a = b) == 0
```

If you do this, be sure you are clear on the difference between the `=` and `==` operators!

The assignment operator has right-to-left associativity, which means that when multiple assignment operators appear in an expression, they are evaluated from right to left. Thus, you can write code like this to assign a single value to multiple variables:

```
i = j = k = 0;
```

Remember that each assignment expression has a value that is the value of the righthand side. So in the above code, the value of the first assignment (the rightmost one) becomes the righthand side for the second assignment (the middle one), and this value becomes the righthand side for the last (leftmost) assignment.

5.9.1 Assignment with Operation

Besides the normal `=` assignment operator, JavaScript supports a number of other assignment operators that provide shortcuts by combining assignment with some other operation. For example, the `+=` operator performs addition and assignment. The following expression:

```
total += sales_tax
```

is equivalent to this one:

```
total = total + sales_tax
```

As you might expect, the `+=` operator works for numbers or strings. For numeric operands, it performs addition and assignment; for string operands, it performs concatenation and assignment.

Similar operators include `-=`, `*=`, `&=`, and so on. [Table 5-2](#) lists them all. In most cases, the expression:

```
a  
op= b
```

where `op` is an operator, is equivalent to the expression:

```
a = a  
op b
```

These expressions differ only if `a` contains side effects such as a function call or an increment operator.

Operator	Example	Equivalent
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>
<code>*=</code>	<code>a *= b</code>	<code>a = a * b</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>%=</code>	<code>a %= b</code>	<code>a = a % b</code>
<code><<=</code>	<code>a <<= b</code>	<code>a = a << b</code>
<code>>>=</code>	<code>a >>= b</code>	<code>a = a >> b</code>
<code>>>>=</code>	<code>a >>>= b</code>	<code>a = a >>> b</code>
<code>&=</code>	<code>a &= b</code>	<code>a = a & b</code>
<code> =</code>	<code>a = b</code>	<code>a = a b</code>
<code>^=</code>	<code>a ^= b</code>	<code>a = a ^ b</code>

5.10 Miscellaneous Operators

JavaScript supports a number of other miscellaneous operators, described in the following sections.

5.10.1 The Conditional Operator (?:)

The conditional operator is the only ternary operator (three operands) in JavaScript and is sometimes actually called the ternary operator. This operator is sometimes written `? :`, although it does not appear quite that way in code. Because this operator has three operands, the first goes before the `?`, the second goes between the `?` and the `:`, and the third goes after the `:`. It is used like this:

```
x > 0 ? x*y : -x*y
```

The first operand of the conditional operator must be (or be convertible to) a boolean value -- usually this is the result of a comparison expression. The second and third operands may have any value. The value returned by the conditional operator depends on

the boolean value of the first operand. If that operand is `true`, the value of the conditional expression is the value of the second operand. If the first operand is `false`, the value of the conditional expression is the value of the third operand.

While you can achieve similar results using the `if` statement, the `?:` operator often provides a handy shortcut. Here is a typical usage, which checks to be sure that a variable is defined, uses it if so, and provides a default value if not:

```
greeting = "hello " + (username != null ? username : "there");
```

This is equivalent to, but more compact than, the following `if` statement:

```
greeting = "hello ";
if (username != null)
    greeting += username;
else
    greeting += "there";
```

5.10.2 The `typeof` Operator

`typeof` is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The `typeof` operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value. It evaluates to "object" for objects, arrays, and (surprisingly) `null`. It evaluates to "function" for function operands and to "undefined" if the operand is undefined.

`typeof` evaluates to "object" when its operand is a Number, String, or Boolean wrapper object. It also evaluates to "object" for Date and RegExp objects. `typeof` evaluates to an implementation-dependent value for objects that are not part of core JavaScript but are provided by the context in which JavaScript is embedded. In client-side JavaScript, however, `typeof` typically evaluates to "object" for all client-side objects, just as it does for all core objects.

You might use the `typeof` operator in expressions like these:

```
typeof i
(typeof value == "string") ? "'" + value + "'" : value
```

Note that you can place parentheses around the operand to `typeof`, which makes `typeof` look like the name of a function rather than an operator keyword:

```
typeof(i)
```

Because `typeof` evaluates to "object" for all object and array types, it is useful only to distinguish objects from other, primitive types. In order to distinguish one object type from another, you must use other techniques, such as the `instanceof` operator or the `constructor` property (see the "Object.constructor" entry in the core reference section).

The `typeof` operator is defined by the ECMAScript v1 specification and is implemented in JavaScript 1.1 and later.

5.10.3 The Object Creation Operator (`new`)

The `new` operator creates a new object and invokes a constructor function to initialize it. `new` is a unary operator that appears before a constructor invocation. It has the following syntax:

```
new constructor(arguments)
```

`constructor` must be an expression that evaluates to a constructor function, and it should be followed by zero or more comma-separated arguments enclosed in parentheses. As a special case, for the `new` operator only, JavaScript simplifies the grammar by allowing the parentheses to be omitted if there are no arguments in the function call. Here are some examples using the `new` operator:

```
o = new Object;      // Optional parentheses omitted here
d = new Date( );    // Returns a Date object representing the current
                    // time
c = new Rectangle(3.0, 4.0, 1.5, 2.75); // Create an object of class
Rectangle
obj[i] = new constructors[i]( );
```

The `new` operator first creates a new object with no properties defined; next, it invokes the specified constructor function, passing the specified arguments and also passing the newly created object as the value of the `this` keyword. The constructor function can then use the `this` keyword to initialize the new object in any way desired. We'll learn more about the `new` operator, the `this` keyword, and constructor functions in [Chapter 8](#).

The `new` operator can also be used to create arrays, using the `new Array()` syntax. We'll see more about creating and working with objects and arrays in [Chapter 8](#) and [Chapter 9](#).

5.10.4 The delete Operator

`delete` is a unary operator that attempts to delete the object property, array element, or variable specified as its operand.^[1] It returns `true` if the deletion was successful, and `false` if the operand could not be deleted. Not all variables and properties can be deleted: some built-in core and client-side properties are immune from deletion, and user-defined variables declared with the `var` statement cannot be deleted. If `delete` is invoked on a

nonexistent property, it returns `true`. (Surprisingly, the ECMAScript standard specifies that `delete` also evaluates to `true` if the operand is not a property, array element, or variable.) Here are some examples of the use of this operator:

^[3] If you are a C++ programmer, note that the `delete` operator in JavaScript is nothing like the `delete` operator in C++. In JavaScript, memory deallocation is handled automatically by garbage collection, and you never have to worry about explicitly freeing up memory. Thus, there is no need for a C++-style `delete` to delete entire objects.

```
var o = {x:1, y:2}; // Define a variable; initialize it to an object
delete o.x;       // Delete one of the object properties; returns
true
typeof o.x;      // Property does not exist; returns "undefined"
delete o.x;      // Delete a nonexistent property; returns true
delete o;        // Can't delete a declared variable; returns false
delete 1;        // Can't delete an integer; returns true
x = 1;           // Implicitly declare a variable without var
keyword
delete x;        // Can delete this kind of variable; returns true
x;              // Runtime error: x is not defined
```

Note that a deleted property, variable, or array element is not merely set to the `undefined` value. When a property is deleted, the property ceases to exist. See the related discussion in [Section 4.3.2](#).

`delete` is standardized by the ECMAScript v1 specification and implemented in JavaScript 1.2 and later. Note that the `delete` operator exists in JavaScript 1.0 and 1.1 but does not actually perform deletion in those versions of the language. Instead, it merely sets the specified property, variable, or array element to `null`.

It is important to understand that `delete` affects only properties, not objects referred to by those properties. Consider the following code:

```
var my = new Object( ); // Create an object named "my"
my.hire = new Date( ); // my.hire refers to a Date object
my.fire = my.hire;     // my.fire refers to the same object
delete my.hire;        // hire property is deleted; returns true
document.write(my.fire); // But my.fire still refers to the Date
object
```

5.10.5 The void Operator

`void` is a unary operator that appears before its single operand, which may be of any type. The purpose of this operator is an unusual one: it discards its operand value and returns `undefined`. The most common use for this operator is in a client-side `javascript:` URL, where it allows you to evaluate an expression for its side effects without the browser displaying the value of the evaluated expression.

For example, you might use the `void` operator in an HTML tag as follows:

```
<a href="javascript:void window.open( );">Open New Window</a>
```

Another use for `void` is to purposely generate the `undefined` value. `void` is specified by ECMAScript v1 and implemented in JavaScript 1.1. The global `undefined` property, however, is specified by ECMAScript v3 and implemented in JavaScript 1.5. Thus, for backward compatibility, you may find it useful to use an expression like `void 0` instead of relying on the `undefined` property.

5.10.6 The Comma Operator (,)

The comma operator is a simple one. It evaluates its left argument, evaluates its right argument, and then returns the value of its right argument. Thus, the following line:

```
i=0, j=1, k=2;
```

evaluates to 2, and is basically equivalent to:

```
i = 0;  
j = 1;  
k = 2;
```

This strange operator is useful only in a few limited circumstances, primarily when you need to evaluate several independent expressions with side effects in a situation where only a single expression is allowed. In practice, the comma operator is really used only in conjunction with the `for` loop statement, which we'll see in [Chapter 6](#).

5.10.7 Array and Object Access Operators

As noted briefly in [Chapter 3](#), you can access elements of an array using square brackets (`[]`), and you can access elements of an object using a dot (`.`). Both `[]` and `.` are treated as operators in JavaScript.

The `.` operator expects an object as its left operand and an identifier (a property name) as its right operand. The right operand should not be a string or a variable that contains a string; it should be the literal name of the property or method, without quotes of any kind. Here are some examples:

```
document.lastModified  
navigator.appName  
frames[0].length  
document.write("hello world")
```

If the specified property does not exist in the object, JavaScript does not issue an error, but instead simply returns `undefined` as the value of the expression.

Most operators allow arbitrary expressions for either operand, as long as the type of the operand is suitable. The `.` operator is an exception: the righthand operand must be an identifier. Nothing else is allowed.

The `[]` operator allows access to array elements. It also allows access to object properties without the restrictions that the `.` operator places on the righthand operand. If the first operand (which goes before the left bracket) refers to an array, the second operand (which goes between the brackets) should be an expression that evaluates to an integer. For example:

```
frames[1]
document.forms[i + j]
document.forms[i].elements[j++]
```

If the first operand to the `[]` operator is a reference to an object, the second operand should be an expression that evaluates to a string that names a property of the object. Note that in this case, the second operand is a string, not an identifier. It should be a constant in quotes or a variable or expression that refers to a string. For example:

```
document["lastModified"]
frames[0]['length']
data["val" + i]
```

The `[]` operator is typically used to access the elements of an array. It is less convenient than the `.` operator for accessing properties of an object because of the need to quote the name of the property. When an object is used as an associative array, however, and the property names are dynamically generated, the `.` operator cannot be used; only the `[]` operator will do. This is commonly the case when you use the `for/in` loop, which is introduced in [Chapter 6](#). For example, the following JavaScript code uses a `for/in` loop and the `[]` operator to print out the names and values of all of the properties in an object `o`:

```
for (f in o) {
    document.write('o.' + f + ' = ' + o[f]);
    document.write('<br>');
}
```

5.10.8 The Function Call Operator

The `()` operator is used to invoke functions in JavaScript. This is an unusual operator in that it does not have a fixed number of operands. The first operand is always the name of a function or an expression that refers to a function. It is followed by the left parenthesis

and any number of additional operands, which may be arbitrary expressions, each separated from the next with a comma. The right parenthesis follows the final operand. The () operator evaluates each of its operands and then invokes the function specified by the first operand, with the values of the remaining operands passed as arguments. For example:

```
document.close( )  
Math.sin(x)  
alert("Welcome " + name)  
Date.UTC(2000, 11, 31, 23, 59, 59)  
funcs[i].f(funcs[i].args[0], funcs[i].args[1])
```

Chapter 6. Statements

As we saw in the last chapter, expressions are JavaScript phrases that can be evaluated to yield a value. Operators within an expression may have side effects, but in general, expressions don't do anything. To make something happen, you use a JavaScript *statement*, which is akin to a complete sentence or command. This chapter describes the various statements in JavaScript and explains their syntax. A JavaScript program is simply a collection of statements, so once you are familiar with the statements of JavaScript, you can begin writing JavaScript programs.

Before we examine JavaScript statements, recall from [Section 2.4](#), that statements in JavaScript are separated from each other with semicolons. If you place each statement on a separate line, however, JavaScript allows you to leave out the semicolons. Nevertheless, it is a good idea to get in the habit of using semicolons everywhere.

6.1 Expression Statements

The simplest kinds of statements in JavaScript are expressions that have side effects. We've seen this sort of statement in [Chapter 5](#). Assignment statements are one major category of expression statements. For example:

```
s = "Hello " + name;  
i *= 3;
```

The increment and decrement operators, ++ and --, are related to assignment statements. These have the side effect of changing a variable value, just as if an assignment had been performed:

```
counter++;
```

The `delete` operator has the important side effect of deleting an object property. Thus, it is almost always used as a statement, rather than as part of a larger expression:

```
delete o.x;
```

Function calls are another major category of expression statements. For example:

```
alert("Welcome, " + name);  
window.close( );
```

These client-side function calls are expressions, but they also affect the web browser, so they are statements, too. If a function does not have any side effects, there is no sense in

calling it, unless it is part of an assignment statement. For example, you wouldn't just compute a cosine and discard the result:

```
Math.cos(x);
```

Instead, you'd compute the value and assign it to a variable for future use:

```
cx = Math.cos(x);
```

Again, please note that each line of code in each of these examples is terminated with a semicolon.

6.2 Compound Statements

In [Chapter 5](#), we saw that the comma operator can be used to combine a number of expressions into a single expression. JavaScript also has a way to combine a number of statements into a single statement, or *statement block*. This is done simply by enclosing any number of statements within curly braces. Thus, the following lines act as a single statement and can be used anywhere that JavaScript expects a single statement:

```
{
  x = Math.PI;
  cx = Math.cos(x);
  alert("cos(" + x + ") = " + cx);
}
```

Note that although this statement block acts as a single statement, it does *not* end with a semicolon. The primitive statements within the block end in semicolons, but the block itself does not.

Although combining expressions with the comma operator is an infrequently used technique, combining statements into larger statement blocks is extremely common. As we'll see in the following sections, a number of JavaScript statements themselves contain statements (just as expressions can contain other expressions); these statements are *compound statements*. Formal JavaScript syntax specifies that each of these compound statements contains a single substatement. Using statement blocks, you can place any number of statements within this single allowed substatement.

To execute a compound statement, the JavaScript interpreter simply executes the statements that comprise it one after another, in the order in which they are written. Normally, the JavaScript interpreter executes all of the statements. In some circumstances, however, a compound statement may terminate abruptly. This termination occurs if the compound statement contains a `break`, `continue`, `return`, or `throw` statement, if it causes an error, or if it calls a function that causes an uncaught error or

throws an uncaught exception. We'll learn more about these abrupt terminations in later sections.

6.3 if

The `if` statement is the fundamental control statement that allows JavaScript to make decisions, or, more precisely, to execute statements conditionally. This statement has two forms. The first is:

```
if (expression)
    statement
```

In this form, *expression* is evaluated. If the resulting value is `true` or can be converted to `true`, *statement* is executed. If *expression* is `false` or converts to `false`, *statement* is not executed. For example:

```
if (username == null)          // If username is null or undefined,
    username = "John Doe";    // define it
```

Or similarly:

```
// If username is null, undefined, 0, "", or NaN, it converts to false,
// and this statement will assign a new value to it.
if (!username) username = "John Doe";
```

Although they look extraneous, the parentheses around the expression are a required part of the syntax for the `if` statement.

As mentioned in the previous section, we can always replace a single statement with a statement block. So the `if` statement might also look like this:

```
if ((address == null) || (address == "")) {
    address = "undefined";
    alert("Please specify a mailing address.");
}
```

The indentation used in these examples is not mandatory. Extra spaces and tabs are ignored in JavaScript, and since we used semicolons after all the primitive statements, these examples could have been written all on one line. Using line breaks and indentation as shown here, however, makes the code easier to read and understand.

The second form of the `if` statement introduces an `else` clause that is executed when *expression* is `false`. Its syntax is:

```
if (expression)
    statement1
else
    statement2
```

In this form of the statement, *expression* is evaluated, and if it is `true`, *statement1* is executed; otherwise, *statement2* is executed. For example:

```
if (username != null)
    alert("Hello " + username + "\nWelcome to my home page.");
else {
    username = prompt("Welcome!\n What is your name?");
    alert("Hello " + username);
}
```

When you have nested `if` statements with `else` clauses, some caution is required to ensure that the `else` clause goes with the appropriate `if` statement. Consider the following lines:

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
        document.write("i equals k");
else
    document.write("i doesn't equal j");    // WRONG!!
```

In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear (except from the hint given by the indentation) which `if` the `else` goes with. And in this example, the indenting hint is wrong, because a JavaScript interpreter actually interprets the previous example as:

```
if (i == j) {
    if (j == k)
        document.write("i equals k");
else
    document.write("i doesn't equal j");    // OOPS!
}
```

The rule in JavaScript (as in most programming languages) is that an `else` clause is part of the nearest `if` statement. To make this example less ambiguous and easier to read, understand, maintain, and debug, you should use curly braces:

```
if (i == j) {
    if (j == k) {
        document.write("i equals k");
    }
}
```

```
}  
else { // What a difference the location of a curly brace makes!  
    document.write("i doesn't equal j");  
}
```

Although it is not the style used in this book, many programmers make a habit of enclosing the bodies of `if` and `else` statements (as well as other compound statements, such as `while` loops) within curly braces, even when the body consists of only a single statement. Doing so consistently can prevent the sort of problem just shown.

6.4 else if

We've seen that the `if/else` statement is useful for testing a condition and executing one of two pieces of code, depending on the outcome. But what about when we need to execute one of many pieces of code? One way to do this is with an `else if` statement. `else if` is not really a JavaScript statement, but simply a frequently used programming idiom that results when repeated `if/else` statements are used:

```
if (n == 1) {  
    // Execute code block #1  
}  
else if (n == 2) {  
    // Execute code block #2  
}  
else if (n == 3) {  
    // Execute code block #3  
}  
else {  
    // If all else fails, execute block #4  
}
```

There is nothing special about this code. It is just a series of `if` statements, where each `if` is part of the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing these statements out in their syntactically equivalent fully nested form:

```
if (n == 1) {  
    // Execute code block #1  
}  
else {  
    if (n == 2) {  
        // Execute code block #2  
    }  
    else {  
        if (n == 3) {  
            // Execute code block #3  
        }  
        else {  
            // If all else fails, execute block #4  
        }  
    }  
}
```

```
    }  
  }  
}
```

6.5 switch

An `if` statement causes a branch in the flow of a program's execution. You can use multiple `if` statements, as in the previous section, to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable. In this case, it is wasteful to repeatedly check the value of the same variable in multiple `if` statements.

The `switch` statement (implemented in JavaScript 1.2 and standardized by ECMAScript v3) handles exactly this situation, and it does so more efficiently than repeated `if` statements. The JavaScript `switch` statement is quite similar to the `switch` statement in Java or C. The `switch` keyword is followed by an expression and a block of code, much like the `if` statement:

```
switch(expression) {  
    statements  
}
```

However, the full syntax of a `switch` statement is more complex than this. Various locations in the block of code are labeled with the `case` keyword followed by a value and a colon. When a `switch` executes, it computes the value of `expression` and then looks for a `case` label that matches that value. If it finds one, it starts executing the block of code at the first statement following the `case` label. If it does not find a `case` label with a matching value, it starts execution at the first statement following a special-case `default:` label. Or, if there is no `default:` label, it skips the block of code altogether.

`switch` is a confusing statement to explain; its operation becomes much clearer with an example. The following `switch` statement is equivalent to the repeated `if/else` statements shown in the previous section:

```
switch(n) {  
    case 1: // Start here if n == 1  
        // Execute code block #1.  
        break; // Stop here  
    case 2: // Start here if n == 2  
        // Execute code block #2.  
        break; // Stop here  
    case 3: // Start here if n == 3  
        // Execute code block #3.  
        break; // Stop here  
    default: // If all else fails...  
        // Execute code block #4.  
        break; // stop here
```

```
}
```

Note the `break` keyword used at the end of each `case` in the code above. The `break` statement, described later in this chapter, causes execution to jump to the end of a `switch` statement or loop. The `case` clauses in a `switch` statement specify only the *starting point* of the desired code; they do not specify any ending point. In the absence of `break` statements, a `switch` statement begins executing its block of code at the `case` label that matches the value of its *expression* and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that falls through from one `case` label to the next, but 99% of the time you should be careful to end every `case` within a `switch` with a `break` statement. (When using `switch` inside a function, however, you may use a `return` statement instead of a `break` statement. Both serve to terminate the `switch` statement and prevent execution from falling through to the next `case`.)

Here is a more realistic example of the `switch` statement; it converts a value to a string in a way that depends on the type of the value:

```
function convert(x) {
  switch(typeof x) {
    case 'number':           // Convert the number to a hexadecimal
integer
    return x.toString(16);
    case 'string':          // Return the string enclosed in quotes
    return '"' + x + '"';
    case 'boolean':         // Convert to TRUE or FALSE, in
uppercase
    return x.toString().toUpperCase( );
  default:                  // Convert any other type in the usual
way
    return x.toString( );
  }
}
```

Note that in the two previous examples, the `case` keywords are followed by number and string literals. This is how the `switch` statement is most often used in practice, but note that the ECMAScript v3 standard allows each `case` to be followed by an arbitrary expression.^[1] For example:

^[1] This makes the JavaScript `switch` statement much different from the `switch` statement of C, C++, and Java. In those languages, the `case` expressions must be compile-time constants, they must evaluate to integers or other integral types, and they must all evaluate to the same type.

```
case 60*60*24:
case Math.PI:
case n+1:
case a[0]:
```


The `switch` statement first evaluates the expression that follows the `switch` keyword, then evaluates the `case` expressions, in the order in which they appear, until it finds a value that matches.^[2] The matching case is determined using the `===` identity operator, not the `==` equality operator, so the expressions must match without any type conversion.

^[2] This means that the JavaScript `switch` statement is not nearly as efficient as the `switch` statement in C, C++, and Java. Since the `case` expressions in those languages are compile-time constants, they never need to be evaluated at runtime as they are in JavaScript. Furthermore, since the case expressions are integral values in C, C++, and Java, the `switch` statement can often be implemented using a highly efficient "jump table."

Note that it is not good programming practice to use `case` expressions that contain side effects such as function calls or assignments, because not all of the `case` expressions are evaluated each time the `switch` statement is executed. When side effects occur only sometimes, it can be difficult to understand and predict the correct behavior of your program. The safest course is simply to limit your `case` expressions to constant expressions.

As explained earlier, if none of the `case` expressions match the `switch` expression, the `switch` statement begins executing its body at the statement labeled `default:`. If there is no `default:` label, the `switch` statement skips its body altogether. Note that in the earlier examples, the `default:` label appears at the end of the `switch` body, following all the `case` labels. This is a logical and common place for it, but it can actually appear anywhere within the body of the statement.

The `switch` statement is implemented in JavaScript 1.2, but it does not fully conform to the ECMAScript specification. In JavaScript 1.2, `case` expressions must be literals or compile-time constants that do not involve any variables or method calls. Furthermore, although ECMAScript allows the `switch` and `case` expressions to be of any type, JavaScript 1.2 and JavaScript 1.3 require that the expressions evaluate to primitive numbers, strings, or boolean values.

6.6 while

Just as the `if` statement is the basic control statement that allows JavaScript to make decisions, the `while` statement is the basic statement that allows JavaScript to perform repetitive actions. It has the following syntax:

```
while (expression)  
    statement
```

The `while` statement works by first evaluating `expression`. If it is `false`, JavaScript moves on to the next statement in the program. If it is `true`, the `statement` that forms the body of the loop is executed and `expression` is evaluated again. Again, if the value of `expression` is `false`, JavaScript moves on to the next statement in the program; otherwise, it executes `statement` again. This cycle continues until `expression` evaluates

to `false`, at which point the `while` statement ends and JavaScript moves on. Note that you can create an infinite loop with the syntax `while(true)`.

Usually, you do not want JavaScript to perform exactly the same operation over and over again. In almost every loop, one or more variables change with each *iteration* of the loop. Since the variables change, the actions performed by executing *statement* may differ each time through the loop. Furthermore, if the changing variable or variables are involved in *expression*, the value of the expression may be different each time through the loop. This is important -- otherwise, an expression that starts off `true` would never change and the loop would never end! Here is an example `while` loop:

```
var count = 0;
while (count < 10) {
    document.write(count + "<br>");
    count++;
}
```

As you can see, the variable `count` starts off at 0 in this example and is incremented each time the body of the loop runs. Once the loop has executed 10 times, the expression becomes `false` (i.e., the variable `count` is no longer less than 10), the `while` statement finishes, and JavaScript can move on to the next statement in the program. Most loops have a counter variable like `count`. The variable names `i`, `j`, and `k` are commonly used as a loop counters, though you should use more descriptive names if it makes your code easier to understand.

6.7 do/while

The `do/while` loop is much like a `while` loop, except that the loop expression is tested at the bottom of the loop rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

```
do
    statement
while (expression);
```

The `do/while` statement is implemented in JavaScript 1.2 and later and standardized by ECMAScript v3.

The `do/while` loop is less commonly used than its `while` cousin. This is because, in practice, it is somewhat uncommon to encounter a situation in which you are always sure that you want a loop to execute at least once. For example:

```
function printArray(a) {
    if (a.length == 0)
        document.write("Empty Array");
    else {
```

```

    var i = 0;
    do {
        document.write(a[i] + "<br>");
    } while (++i < a.length);
}

```

There are a couple of differences between the `do/while` loop and the ordinary `while` loop. First, the `do` loop requires both the `do` keyword (to mark the beginning of the loop) and the `while` keyword (to mark the end and introduce the loop condition). Also, unlike the `while` loop, the `do` loop is terminated with a semicolon. This is because the `do` loop ends with the loop condition, rather than simply with a curly brace that marks the end of the loop body.

In JavaScript 1.2, there is a bug in the behavior of the `continue` statement (see [Section 6.12](#)) when it is used inside a `do/while` loop. For this reason, you should avoid the use of `continue` within `do/while` statements in JavaScript 1.2.

6.8 for

The `for` statement provides a looping construct that is often more convenient than the `while` statement. The `for` statement takes advantage of a pattern common to most loops (including the earlier `while` loop example). Most loops have a counter variable of some kind. This variable is initialized before the loop starts and is tested as part of the *expression* evaluated before each iteration of the loop. Finally, the counter variable is incremented or otherwise updated at the end of the loop body, just before *expression* is evaluated again.

The initialization, the test, and the update are the three crucial manipulations of a loop variable; the `for` statement makes these three steps an explicit part of the loop syntax. This makes it especially easy to understand what a `for` loop is doing and prevents mistakes such as forgetting to initialize or increment the loop variable. The syntax of the `for` statement is:

```

for(initialize ; test ; increment)
    statement

```

The simplest way to explain what this `for` loop does is to show the equivalent `while` loop:^[3]

^[3] As we will see when we consider the `continue` statement, this `while` loop is not an exact equivalent to the `for` loop.

```

initialize;
while(test) {
    statement
    increment;
}

```

```
}
```

In other words, the *initialize* expression is evaluated once, before the loop begins. To be useful, this is an expression with side effects (usually an assignment). JavaScript also allows *initialize* to be a *var* variable declaration statement, so that you can declare and initialize a loop counter at the same time. The *test* expression is evaluated before each iteration and controls whether the body of the loop is executed. If the *test* expression is *true*, the *statement* that is the body of the loop is executed. Finally, the *increment* expression is evaluated. Again, this must be an expression with side effects in order to be useful. Generally, either it is an assignment expression or it uses the *++* or *--* operators.

The example *while* loop of the previous section can be rewritten as the following *for* loop, which counts from 0 to 9:

```
for(var count = 0 ; count < 10 ; count++)  
    document.write(count + "<br>");
```

Notice that this syntax places all the important information about the loop variable on a single line, which makes it clear how the loop executes. Also note that placing the *increment* expression in the *for* statement itself simplifies the body of the loop to a single statement; we don't even need to use curly braces to produce a statement block.

Loops can become a lot more complex than these simple examples, of course, and sometimes multiple variables change with each iteration of the loop. This situation is the only place that the comma operator is commonly used in JavaScript -- it provides a way to combine multiple initialization and increment expressions into a single expression suitable for use in a *for* loop. For example:

```
for(i = 0, j = 10 ; i < 10 ; i++, j--)  
    sum += i * j;
```

6.9 for/in

The *for* keyword is used in two ways in JavaScript. We've just seen how it is used in the *for* loop. It is also used in the *for/in* statement. This statement is a somewhat different kind of loop with the following syntax:

```
for (variable in object)  
    statement
```

variable should be either the name of a variable, a *var* statement declaring a variable, an element of an array, or a property of an object (i.e., it should be something suitable as the left-hand side of an assignment expression). *object* is the name of an object or an

expression that evaluates to an object. As usual, *statement* is the statement or statement block that forms the body of the loop.

You can loop through the elements of an array by simply incrementing an index variable each time through a `while` or `for` loop. The `for/in` statement provides a way to loop through the properties of an object. The body of the `for/in` loop is executed once for each property of *object*. Before the body of the loop is executed, the name of one of the object's properties is assigned to *variable*, as a string. Within the body of the loop, you can use this variable to look up the value of the object's property with the `[]` operator. For example, the following `for/in` loop prints the name and value of each property of an object:

```
for (var prop in my_object) {
    document.write("name: " + prop + "; value: " + my_object[prop],
"<br>");
}
```

Note that the *variable* in the `for/in` loop may be an arbitrary expression, as long as it evaluates to something suitable for the lefthand side of an assignment. This expression is evaluated each time through the loop, which means that it may evaluate differently each time. For example, you could use code like the following to copy the names of all object properties into an array:

```
var o = {x:1, y:2, z:3};
var a = new Array( );
var i = 0;
for(a[i++] in o) /* empty loop body */;
```

JavaScript arrays are simply a specialized kind of object. Therefore, the `for/in` loop enumerates array indexes as well as object properties. For example, following the previous code block with this line enumerates the array "properties" 0, 1, and 2:

```
for(i in a) alert(i);
```

The `for/in` loop does not specify the order in which the properties of an object are assigned to the variable. There is no way to tell what the order will be in advance, and the behavior may differ between implementations or versions of JavaScript. If the body of a `for/in` loop deletes a property that has not yet been enumerated, that property will not be enumerated. If the body of the loop defines new properties, whether or not those properties will be enumerated by the loop is implementation-dependent.

The `for/in` loop does not actually loop through all possible properties of all objects. In the same way that some object properties are flagged to be read-only or permanent (nondeletable), certain properties are flagged to be nonenumerable. These properties are not enumerated by the `for/in` loop. While all user-defined properties are enumerated,

many built-in properties, including all built-in methods, are not enumerated. As we'll see in [Chapter 8](#), objects can inherit properties from other objects. Inherited properties that are user-defined are also enumerated by the `for/in` loop.

6.10 Labels

The `case` and `default`: labels used in conjunction with the `switch` statement are a special case of a more general label statement. In JavaScript 1.2, any statement may be labeled by preceding it with an identifier name and a colon:

```
identifier: statement
```

The *identifier* can be any legal JavaScript identifier that is not a reserved word. Label names are distinct from variable and function names, so you do not need to worry about name collisions if you give a label the same name as a variable or function. Here is an example of a labeled `while` statement:

```
parser:
  while(token != null) {
    // Code omitted here
  }
```

By labeling a statement, you give it a name that you can use to refer to it elsewhere in your program. You can label any statement, although the only statements that are commonly labeled are loops: `while`, `do/while`, `for`, and `for/in`. By giving a loop a name, you can use `break` and `continue` to exit the loop or to exit a single iteration of the loop.

6.11 break

The `break` statement causes the innermost enclosing loop or a `switch` statement to exit immediately. Its syntax is simple:

```
break;
```

Because it causes a loop or `switch` to exit, this form of the `break` statement is legal only if it appears within one of these statements.

ECMAScript v3 and JavaScript 1.2 allow the `break` keyword to be followed by the name of a label:

```
break labelname;
```

Note that *labelname* is simply an identifier; it is not followed by a colon, as it would be when defining a labeled statement.

When `break` is used with a label, it jumps to the end of, or terminates, the named statement, which may be any enclosing statement. The named statement need not be a loop or `switch`; a `break` statement used with a label need not even be contained within a loop or `switch`. The only restriction on the label of the `break` statement is that it name an *enclosing* statement. The label can name an `if` statement, for example, or even a block of statements grouped within curly braces, for the sole purpose of naming the block with a label.

As discussed in [Chapter 2](#), a newline is not allowed between the `break` keyword and the *labelname*. This is an oddity of JavaScript syntax caused by its automatic insertion of omitted semicolons. If you break a line of code between the `break` keyword and the following label, JavaScript assumes you meant to use the simple, unlabeled form of the statement and adds a semicolon for you.

We've already seen examples of the `break` statement within a `switch` statement. In loops, it is typically used to exit prematurely when, for whatever reason, there is no longer any need to complete the loop. When a loop has complex termination conditions, it is often easier to implement some of these conditions with `break` statements, rather than trying to express them all in a single loop expression.

The following code searches the elements of an array for a particular value. The loop terminates naturally when it reaches the end of the array; it terminates with a `break` statement if it finds what it is looking for in the array:

```
for(i = 0; i < a.length; i++) {
    if (a[i] == target)
        break;
}
```

You need the labeled form of the `break` statement only when you are using nested loops or `switch` statements and need to break out of a statement that is not the innermost one.

The following example shows labeled `for` loops and labeled `break` statements. See if you can figure out what its output will be:

```
outerloop:
  for(var i = 0; i < 10; i++) {
    innerloop:
      for(var j = 0; j < 10; j++) {
        if (j > 3) break; // Quit the innermost loop
        if (i == 2) break innerloop; // Do the same thing
        if (i == 4) break outerloop; // Quit the outer loop
        document.write("i = " + i + " j = " + j + "<br>");
      }
  }
```

```
}  
document.write("FINAL i = " + i + " j = " + j + "<br>");
```

6.12 continue

The `continue` statement is similar to the `break` statement. Instead of exiting a loop, however, `continue` restarts a loop in a new iteration. The `continue` statement's syntax is just as simple as the `break` statement's:

```
continue;
```

In ECMAScript v3 and JavaScript 1.2, the `continue` statement can also be used with a label:

```
continue labelname;
```

The `continue` statement, in both its labeled and unlabeled forms, can be used only within the body of a `while`, `do/while`, `for`, or `for/in` loop. Using it anywhere else causes a syntax error.

When the `continue` statement is executed, the current iteration of the enclosing loop is terminated and the next iteration begins. This means different things for different types of loops:

- In a `while` loop, the specified *expression* at the beginning of the loop is tested again, and if it's `true`, the loop body is executed starting from the top.
- In a `do/while` loop, execution skips to the bottom of the loop, where the loop condition is tested again before restarting the loop at the top. Note, however, that JavaScript 1.2 contains a bug that causes the `continue` statement to jump directly to the top of a `do/while` loop without testing the loop condition. Therefore, if you plan to use a `continue` statement in a loop, you should avoid the `do/while` loop. This is not a serious problem, however, because you can always replace a `do/while` loop with an equivalent `while` loop.
- In a `for` loop, the *increment* expression is evaluated and the *test* expression is tested again to determine if another iteration should be done.
- In a `for/in` loop, the loop starts over with the next property name being assigned to the specified variable.

Note the difference in behavior of the `continue` statement in the `while` and `for` loops -- a `while` loop returns directly to its condition, but a `for` loop first evaluates its *increment* expression and then returns to its condition. Previously, in the discussion of the `for` loop, I explained the behavior of the `for` loop in terms of an equivalent `while` loop. Because the `continue` statement behaves differently for these two loops, it is not possible to perfectly simulate a `for` loop with a `while` loop.

The following example shows an unlabeled `continue` statement being used to exit the current iteration of a loop when an error occurs:

```
for(i = 0; i < data.length; i++) {
  if (data[i] == null)
    continue; // Can't proceed with undefined data
  total += data[i];
}
```

Like the `break` statement, the `continue` statement can be used in its labeled form within nested loops, when the loop to be restarted is not the immediately enclosing loop. Also, like the `break` statement, line breaks are not allowed between the `continue` statement and its *labelname*.

6.13 var

The `var` statement allows you to explicitly declare a variable or variables. The syntax of this statement is:

```
var name_1 [ = value_1] [ , ..., name_n [= value_n]]
```

The `var` keyword is followed by a comma-separated list of variables to declare; each variable in the list may optionally have an initializer expression that specifies its initial value. For example:

```
var i;
var j = 0;
var p, q;
var greeting = "hello" + name;
var x = 2.34, y = Math.cos(0.75), r, theta;
```

The `var` statement defines each named variable by creating a property with that name either in the call object of the enclosing function or, if the declaration does not appear within a function body, in the global object. The property or properties created by a `var` statement cannot be deleted with the `delete` operator. Note that enclosing a `var` statement in a `with` statement (see [Section 6.18](#)) does not change its behavior.

If no initial value is specified for a variable with the `var` statement, the variable is defined but its initial value is `undefined`.

Note that the `var` statement can also appear as part of the `for` and `for/in` loops. For example:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");
```

```
for(var i in o) document.write(i, "<br>");
```

[Chapter 4](#) contains much more information on JavaScript variables and variable declarations.

6.14 function

The `function` statement defines a JavaScript function. It has the following syntax:

```
function funcname([arg1 [,arg2 [..., argn]]) {  
    statements  
}
```

funcname is the name of the function being defined. This must be an identifier, not a string or an expression. The function name is followed by a comma-separated list of argument names in parentheses. These identifiers can be used within the body of the function to refer to the argument values passed when the function is invoked.

The body of the function is composed of any number of JavaScript statements, contained within curly braces. These statements are not executed when the function is defined. Instead, they are compiled and associated with the new function object for execution when the function is invoked with the `()` function call operator. Note that the curly braces are a required part of the `function` statement. Unlike statement blocks used with `while` loops and other statements, a function body requires curly braces, even if the body consists of only a single statement.

A function definition creates a new function object and stores that object in a newly created property named *funcname*. Here are some example function definitions:

```
function welcome( ) { alert("Welcome to my home page!"); }  
  
function print(msg) {  
    document.write(msg, "<br>");  
}  
  
function hypotenuse(x, y) {  
    return Math.sqrt(x*x + y*y); // return is documented below  
}  
  
function factorial(n) { // A recursive function  
    if (n <= 1) return 1;  
    return n * factorial(n - 1);  
}
```

Function definitions usually appear in top-level JavaScript code. They may also be nested within other function definitions, but only at the "top level" of those functions; that is,

function definitions may not appear within `if` statements, `while` loops, or any other statements.

Technically speaking, the `function` statement is not a statement. Statements cause dynamic behavior in a JavaScript program, while function definitions describe the static structure of a program. Statements are executed at runtime, but functions are defined when JavaScript code is parsed, or compiled, before it is actually run. When the JavaScript parser encounters a function definition, it parses and stores (without executing) the statements that comprise the body of the function. Then it defines a property (in the call object if the function definition is nested in another function; otherwise, in the global object) with the same name as the function to hold the function.

The fact that function definitions occur at parse time rather than at runtime causes some surprising effects. Consider the following code:

```
alert(f(4));    // Displays 16. f( ) can be called before it is
defined.
var f = 0;     // This statement overwrites the property f.
function f(x) { // This "statement" defines the function f before
either
    return x*x; // of the lines above are executed.
}
alert(f);     // Displays 0. f( ) has been overwritten by the
variable f.
```

These unusual results occur because function definition occurs at a different time than variable definition. Fortunately, these situations do not arise very often.

We'll learn more about functions in [Chapter 7](#).

6.15 return

As you'll recall, invoking a function with the `()` operator is an expression. All expressions have values; the `return` statement is used to specify the value returned by a function. This value is the value of the function invocation expression. The syntax of the `return` statement is:

```
return expression;
```

A `return` statement may appear only within the body of a function. It is a syntax error for it to appear anywhere else. When the `return` statement is executed, `expression` is evaluated and returned as the value of the function. Execution of the function stops when the `return` statement is executed, even if there are other statements remaining in the function body. The `return` statement can be used to return a value like this:

```
function square(x) { return x*x; }
```

The `return` statement may also be used without an *expression* to simply terminate execution of the function without returning a value. For example:

```
function display_object(obj) {
    // First make sure our argument is valid
    // Skip the rest of the function if it is not
    if (obj == null) return;
    // Rest of function goes here...
}
```

If a function executes a `return` statement with no *expression*, or if it returns because it reaches the end of the function body, the value of the function call expression is `undefined`.

Because of JavaScript's automatic semicolon insertion, you may not include a line break between the `return` keyword and the expression that follows it.

6.16 throw

An *exception* is a signal that indicates that some sort of exceptional condition or error has occurred. To *throw* an exception is to signal such an error or exceptional condition. To *catch* an exception is to handle it -- to take whatever actions are necessary or appropriate to recover from the exception. In JavaScript, exceptions are thrown whenever a runtime error occurs and whenever the program explicitly throws one using the `throw` statement. Exceptions are caught with the `try/catch/finally` statement, which is described in the next section.^[4]

^[4] The JavaScript `throw` and `try/catch/finally` statements are similar to but not exactly the same as the corresponding statements in C++ and Java.

The `throw` statement has the following syntax:

```
throw expression;
```

expression may evaluate to a value of any type. Commonly, however, it is an `Error` object or an instance of one of the subclasses of `Error`. It can also be useful to throw a string that contains an error message, or a numeric value that represents some sort of error code. Here is some example code that uses the `throw` statement to throw an exception:

```
function factorial(x) {
    // If the input argument is invalid, throw an exception!
    if (x < 0) throw new Error("x must not be negative");
    // Otherwise, compute a value and return normally
}
```

```
    for(var f = 1; x > 1; f *= x, x--) /* empty */ ;
    return f;
}
```

When an exception is thrown, the JavaScript interpreter immediately stops normal program execution and jumps to the nearest exception handler. Exception handlers are written using the `catch` clause of the `try/catch/finally` statement, which is described in the next section. If the block of code in which the exception was thrown does not have an associated `catch` clause, the interpreter checks the next highest enclosing block of code to see if it has an exception handler associated with it. This continues until a handler is found. If an exception is thrown in a function that does not contain a `try/catch/finally` statement to handle it, the exception propagates up to the code that invoked the function. In this way, exceptions propagate up through the lexical structure of JavaScript methods and up the call stack. If no exception handler is ever found, the exception is treated as an error and is reported to the user.

The `throw` statement is standardized by ECMAScript v3 and implemented in JavaScript 1.4. The `Error` class and its subclasses are also part of ECMAScript v3, but they are not implemented until JavaScript 1.5.

6.17 try/catch/finally

The `try/catch/finally` statement is JavaScript's exception-handling mechanism. The `try` clause of this statement simply defines the block of code whose exceptions are to be handled. The `try` block is followed by a `catch` clause, which is a block of statements that are invoked when an exception occurs anywhere within the `try` block. The `catch` clause is followed by a `finally` block containing cleanup code that is guaranteed to be executed, regardless of what happens in the `try` block. Both the `catch` and `finally` blocks are optional, but a `try` block must be accompanied by at least one of these blocks. The `try`, `catch`, and `finally` blocks all begin and end with curly braces. These are a required part of the syntax and cannot be omitted, even if the clause contains only a single statement. Like the `throw` statement, the `try/catch/finally` statement is standardized by ECMAScript v3 and implemented in JavaScript 1.4.

The following code illustrates the syntax and purpose of the `try/catch/finally` statement. In particular, note that the `catch` keyword is followed by an identifier in parentheses. This identifier is like a function argument. It names a local variable that exists only within the body of the `catch` block. JavaScript assigns whatever exception object or value was thrown to this variable:

```
try {
    // Normally, this code runs from the top of the block to the bottom
    // without problems. But it can sometimes throw an exception,
    // either directly, with a throw statement, or indirectly, by calling
    // a method that throws an exception.
}
```

```

catch (e) {
    // The statements in this block are executed if, and only if, the try
    // block throws an exception. These statements can use the local
variable
    // e to refer to the Error object or other value that was thrown.
    // This block may handle the exception somehow, or it may ignore the
    // exception by doing nothing, or it may rethrow the exception with
throw.
}
finally {
    // This block contains statements that are always executed,
regardless of
    // what happens in the try block. They are executed whether the try
    // block terminates:
    // 1) normally, after reaching the bottom of the block
    // 2) because of a break, continue, or return statement
    // 3) with an exception that is handled by a catch clause above
    // 4) with an uncaught exception that is still propagating
}

```

Here is a more realistic example of the `try/catch` statement. It uses the `factorial()` method defined in the previous section and the client-side JavaScript methods `prompt()` and `alert()` for input and output:

```

try {
    // Ask the user to enter a number
    var n = prompt("Please enter a positive integer", "");
    // Compute the factorial of the number, assuming that the user's
    // input is valid
    var f = factorial(n);
    // Display the result
    alert(n + "! = " + f);
}
catch (ex) { // If the user's input was not valid, we end up here
    // Tell the user what the error is
    alert(ex);
}

```

This example is a `try/catch` statement with no `finally` clause. Although `finally` is not used as often as `catch`, it can often be useful. However, its behavior requires additional explanation. The `finally` clause is guaranteed to be executed if any portion of the `try` block is executed, regardless of how the code in the `try` block completes. It is generally used to clean up after the code in the `try` clause.

In the normal case, control reaches the end of the `try` block and then proceeds to the `finally` block, which performs any necessary cleanup. If control leaves the `try` block because of a `return`, `continue`, or `break` statement, the `finally` block is executed before control transfers to its new destination.

If an exception occurs in the `try` block and there is an associated `catch` block to handle the exception, control transfers first to the `catch` block and then to the `finally` block. If there is no local `catch` block to handle the exception, control transfers first to the `finally` block and then propagates up to the nearest containing `catch` clause that can handle the exception.

If a `finally` block itself transfers control with a `return`, `continue`, `break`, or `throw` statement, or by calling a method that throws an exception, the pending control transfer is abandoned and this new transfer is processed. For example, if a `finally` clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a `finally` clause issues a `return` statement, the method returns normally, even if an exception has been thrown and has not yet been handled.

`try` and `finally` can be used together without a `catch` clause. In this case, the `finally` block is simply cleanup code that is guaranteed to be executed, regardless of any `break`, `continue`, or `return` statements within the `try` clause. For example, the following code uses a `try/finally` statement to ensure that a loop counter variable is incremented at the end of each iteration, even when an iteration terminates abruptly because of a `continue` statement:

```
var i = 0, total = 0;
while(i < a.length) {
  try {
    if ((typeof a[i] != "number") || isNaN(a[i])) // If it is not a
number,
        continue; // go on to the next iteration of the loop.
    total += a[i]; // Otherwise, add the number to the total.
  }
  finally {
    i++; // Always increment i, even if we used continue above.
  }
}
```

6.18 with

In [Chapter 4](#), we discussed variable scope and the scope chain -- a list of objects that are searched in order, to perform variable name resolution. The `with` statement is used to temporarily modify the scope chain. It has the following syntax:

```
with (object)
  statement
```

This statement effectively adds `object` to the front of the scope chain, executes `statement`, and then restores the scope chain to its original state.

In practice, you can use the `with` statement to save yourself a lot of typing. In client-side JavaScript, for example, it is common to work with deeply nested object hierarchies. For

instance, you may have to type expressions like this one to access elements of an HTML form:

```
frames[1].document.forms[0].address.value
```

If you need to access this form a number of times, you can use the `with` statement to add the form to the scope chain:

```
with(frames[1].document.forms[0]) {  
    // Access form elements directly here. For example:  
    name.value = "";  
    address.value = "";  
    email.value = "";  
}
```

This reduces the amount of typing you have to do -- you no longer need to prefix each form property name with `frames[1].document.forms[0]`. That object is temporarily part of the scope chain and is automatically searched when JavaScript needs to resolve an identifier like `address`.

Despite its occasional convenience, the use of the `with` statement is frowned upon. JavaScript code that uses `with` is difficult to optimize and may therefore run more slowly than the equivalent code written without the `with` statement. Furthermore, function definitions and variable initializations within the body of a `with` statement can have surprising and counterintuitive behavior.^[5] For these reasons, it is recommended that you avoid the `with` statement.

^[5] This behavior, and the reasons behind it, are too complicated to explain here.

Note that there are other, perfectly legitimate ways to save yourself typing. For instance, we could rewrite the previous example as follows:

```
var form = frames[1].document.forms[0];  
form.name.value = "";  
form.address.value = "";  
form.email.value = "";
```

6.19 The Empty Statement

One final legal statement in JavaScript is the empty statement. It looks like this:

```
;
```

Executing the empty statement obviously has no effect and performs no action. You might think there would be little reason to ever use such a statement, but the empty

statement is occasionally useful when you want to create a loop that has an empty body. For example:

```
// Initialize an array a
for(i=0; i < a.length; a[i++] = 0) ;
```

Note that the accidental inclusion of a semicolon after the right parenthesis of a `for` loop, `while` loop, or `if` statement can cause frustrating bugs that are difficult to detect. For example, the following code probably does not do what the author intended:

```
if ((a == 0) || (b == 0)); // Oops! This line does nothing...
    o = null; // and this line is always executed.
```

When you intentionally use the empty statement, it is a good idea to comment your code in a way that makes it clear that you are doing it on purpose. For example:

```
for(i=0; i < a.length; a[i++] = 0) /* Empty */ ;
```

6.20 Summary of JavaScript Statements

This chapter introduced each of the statements of the JavaScript language. [Table 6-1](#) summarizes these statements, listing the syntax and purpose of each.

Table 6-1. JavaScript statement syntax		
Statement	Syntax	Purpose
<code>break</code>	<code>break;</code> <code>break labelname;</code>	Exit from the innermost loop or <code>switch</code> statement or from the statement named by <code>label</code> .
<code>case</code>	<code>case expression:</code>	Label a statement within a <code>switch</code> statement.
<code>continue</code>	<code>continue;</code> <code>continue labelname;</code>	Restart the innermost loop or the loop named by <code>label</code> .
<code>default</code>	<code>default:</code>	Label the default statement within a <code>switch</code> statement.
<code>do/while</code>	<code>do</code> <code>statement</code> <code>while (expression);</code>	An alternative to the <code>while</code> loop.
<code>empty</code>	<code>;</code>	Do nothing.

Table 6-1. JavaScript statement syntax

Statement	Syntax	Purpose
for	<pre>for (initialize ; test ; increment) statement</pre>	An easy-to-use loop.
for/in	<pre>for (variable in object) statement</pre>	Loop through the properties of an object.
function	<pre>function funcname([arg1[... , argn]]) { statements }</pre>	Declare a function.
if/else	<pre>if (expression) statement1 [else statement2]</pre>	Conditionally execute code.
label	<pre>identifier: statement</pre>	Give <i>statement</i> the name <i>identifier</i> .
return	<pre>return [expression];</pre>	Return from a function or return the value of <i>expression</i> from a function.
switch	<pre>switch (expression) { statements }</pre>	Multiway branch to statements labeled with <code>case</code> or <code>default</code> : .
throw	<pre>throw expression;</pre>	Throw an exception.
try	<pre>try { statements } catch (identifier) { statements } finally { statements }</pre>	Catch an exception.
var	<pre>var name_1 [= value_1] [..., name_n [= value_n]];</pre>	Declare and initialize variables.

Table 6-1. JavaScript statement syntax

Statement	Syntax	Purpose
while	while (<i>expression</i>) <i>statement</i>	A basic loop construct.
with	with (<i>object</i>) <i>statement</i>	Extend the scope chain. (Deprecated.)

Chapter 7. Functions

Functions are an important and complex part of the JavaScript language. This chapter examines functions from several points of view. First, we discuss functions from the syntactic standpoint, explaining how they are defined and invoked. Second, we cover functions as a data type, with examples of the useful programming techniques that are made possible by treating functions as data. Finally, we consider the topic of variable scope within the body of a function and examine some of the useful function-related properties that are available to an executing function. This includes a discussion of how to write JavaScript functions that accept an arbitrary number of arguments.

This chapter focuses on defining and invoking user-defined JavaScript functions. It is also important to remember that JavaScript supports quite a few built-in functions, such as `eval()`, `parseInt()`, and the `sort()` method of the Array class. Client-side JavaScript defines others, such as `document.write()` and `alert()`. Built-in functions in JavaScript can be used in exactly the same ways as user-defined functions. You can find more information about the built-in functions mentioned here in the core and client-side reference sections of this book.

Functions and objects are intertwined in JavaScript. For this reason, I'll defer discussion of some features of functions until [Chapter 8](#).

.1 Defining and Invoking Functions

As we saw in [Chapter 6](#), the most common way to define a function is with the `function` statement. This statement consists of the `function` keyword, followed by:

- The name of the function
- An optional comma-separated list of parameter names in parentheses
- The JavaScript statements that comprise the body of the function, contained within curly braces

[Example 7-1](#) shows the definitions of several functions. Although these functions are short and simple, they all contain each of the elements I just listed. Note that functions may be defined to expect varying numbers of arguments and that they may or may not contain a `return` statement. The `return` statement was introduced in [Chapter 6](#); it causes the function to stop executing and to return the value of its expression (if any) to the caller. If a function does not contain a `return` statement, it simply executes each statement in the function body and returns the `undefined` value to the caller.

Example 7-1. Defining JavaScript functions

```
// A shortcut function, sometimes useful instead of document.write( )
// This function has no return statement, so it returns no value
function print(msg)
{
```

```

    document.write(msg, "<br>");
}
// A function that computes and returns the distance between two points
function distance(x1, y1, x2, y2)
{
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}
// A recursive function (one that calls itself) that computes
factorials
// Recall that x! is the product of x and all positive integers less
than it
function factorial(x)
{
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}

```

Once a function has been defined, it may be invoked with the () operator, introduced in [Chapter 5](#). Recall that the parentheses appear after the name of the function and that an optional comma-separated list of argument values (or expressions) appears within the parentheses. The functions defined in [Example 7-1](#) could be invoked with code like the following:

```

print("Hello, " + name);
print("Welcome to my home page!");
total_dist = distance(0,0,2,1) + distance(2,1,3,5);
print("The probability of that is: " + factorial(39)/factorial(52));

```

When you invoke a function, each of the expressions you specify between the parentheses is evaluated and the resulting value is used as an argument of the function. These values are assigned to the parameters named when the function was defined, and the function operates on its parameters by referring to them by name. Note that these parameter variables are defined only while the function is being executed; they do not persist once the function returns.

Since JavaScript is an untyped language, you are not expected to specify a data type for function parameters, and JavaScript does not check whether you have passed the type of data that the function expects. If the data type of an argument is important, you can test it yourself with the `typeof` operator. JavaScript does not check whether you have passed the correct number of arguments, either. If you pass more arguments than the function expects, the extra values are simply ignored. If you pass fewer than expected, some of the parameters are given the `undefined` value -- which, in many circumstances, causes your function to behave incorrectly. Later in this chapter, we'll see a technique you can use to test whether the correct number of arguments have been passed to a function.

Note that the `print()` function does not contain a `return` statement, so it always returns the `undefined` value and cannot meaningfully be used as part of a larger expression. The `distance()` and `factorial()` functions, on the other hand, can be invoked as parts of larger expressions, as was shown in the previous examples.

7.1.1 Nested Functions

ECMAScript v1 and implementations prior to JavaScript 1.2 allow functions to be defined only in top-level global code. JavaScript 1.2 and ECMAScript v3, however, allow function definitions to be nested within other functions. For example:

```
function hypotenuse(a, b) {  
    function square(x) { return x*x; }  
    return Math.sqrt(square(a) + square(b));  
}
```

Note that ECMAScript v3 does not allow function definitions to appear anywhere; they are still restricted to top-level global code and top-level function code. This means that function definitions may not appear within loops or conditionals, for example.^[1] These restrictions on function definitions apply only to function declarations with the `function` statement. As we'll discuss later in this chapter, function literals (another feature introduced in JavaScript 1.2 and standardized by ECMAScript v3) may appear within any JavaScript expression, which means that they can appear within `if` and other statements.

^[1] Different implementations of JavaScript may be more relaxed about function definitions than the standard requires. For example, Netscape's implementation of JavaScript 1.5 allows "conditional function definitions" that appear within `if` statements.

7.1.2 The Function() Constructor

The `function` statement is not the only way to define a new function. ECMAScript v1 and JavaScript 1.1 allow you to define a function dynamically with the `Function()` constructor and the `new` operator. (We saw the `new` operator in [Chapter 5](#), and we'll learn more about constructors in [Chapter 8](#).) Here is an example of creating a function in this way:

```
var f = new Function("x", "y", "return x*y;");
```

This line of code creates a new function that is more or less equivalent to a function defined with the familiar syntax:

```
function f(x, y) { return x*y; }
```

The `Function()` constructor expects any number of string arguments. The last argument is the body of the function -- it can contain arbitrary JavaScript statements,

separated from each other by semicolons. All other arguments to the constructor are strings that specify the names of the parameters to the function being defined. If you are defining a function that takes no arguments, you simply pass a single string -- the function body -- to the constructor.

Notice that the `Function()` constructor is not passed any argument that specifies a name for the function it creates. The unnamed functions created with the `Function()` constructor are sometimes called *anonymous functions*.

You might well wonder what the point of the `Function()` constructor is. Why not simply define all functions with the `function` statement? One reason is that `Function()` allows us to dynamically build and compile functions; it does not restrict us to the precompiled function bodies of the `function` statement. The flip side of this benefit is that the `Function()` constructor has to compile a function each time it is called. Therefore, you probably do not want to call this constructor within the body of a loop or within a frequently used function.

Another reason to use the `Function()` constructor is that it is sometimes convenient, and even elegant, to be able to define a function as part of a JavaScript expression, rather than as a statement. We'll see examples of this usage later in this chapter. In JavaScript 1.2, when you want to define a function in an expression rather than a statement, a function literal is an even more elegant choice than the `Function()` constructor. We'll consider function literals next.

7.1.3 Function Literals

ECMAScript v3 defines and JavaScript 1.2 implements function literals, which are a third way to create functions. As discussed in [Chapter 3](#), a function literal is an expression that defines an unnamed function. The syntax for a function literal is much like that of the `function` statement, except that it is used as an expression rather than as a statement and no function name is required. The following three lines of code define three more or less identical functions using the `function` statement, the `Function()` constructor, and a function literal:

```
function f(x) { return x*x; }           // function statement
var f = new Function("x", "return x*x;"); // Function( ) constructor
var f = function(x) { return x*x; };   // function literal
```

Although function literals create unnamed functions, the syntax allows a function name to be optionally specified, which is useful when writing recursive functions that call themselves. For example:

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };
```

This line of code defines an unnamed function and stores a reference to it in the variable `f`. It does not actually create a function named `fact()`, but it does allow the body of the function to refer to itself using that name. Note, however, that this type of named function literal is not properly implemented before JavaScript 1.5.

Keep in mind that the `function` statement is available in all versions of JavaScript, the `Function()` constructor is available only in JavaScript 1.1 and later, and function literals are available only in JavaScript 1.2 and later. Recall that we said the three functions defined earlier are "more or less" equivalent -- there are some differences between these three techniques for function definition, which we'll consider in [Section 11.5](#).

Function literals are useful in much the same way as functions created with the `Function()` constructor. Because they are created by JavaScript expressions rather than statements, they can be used in more flexible ways and are particularly suited for functions that are used only once and need not be named. For example, the function specified by a function literal expression can be stored into a variable, passed to another function, or even invoked directly:

```
a[0] = function(x) { return x*x; }; // Define a function and store it
a.sort(function(a,b){return a-b;}); // Define a function; pass it to
another
var tensquared = (function(x) {return x*x;})(10); // Define and invoke
```

Like the `Function()` constructor, function literals create unnamed functions and do not automatically store those functions into properties. Function literals have an important advantage over the `Function()` constructor, however. The body of a function created by `Function()` must be specified in a string, and it can be awkward to express long, complex function bodies in this way. The body of a function literal, however, uses standard JavaScript syntax. Also, a function literal is parsed and compiled only once, while the JavaScript code passed as a string to the `Function()` constructor must be parsed and compiled each time the constructor is invoked.

7.2 Functions as Data

The most important features of functions are that they can be defined and invoked, as shown in the previous section. Function definition and invocation are syntactic features of JavaScript and of most other programming languages. In JavaScript, however, functions are not only syntax but also data, which means that they can be assigned to variables, stored in the properties of objects or the elements of arrays, passed as arguments to functions, and so on.^[2]

^[2] This may not seem like a particularly interesting point to you unless you are familiar with languages like Java, in which functions are part of a program but cannot be manipulated by the program.

To understand how functions can be JavaScript data as well as JavaScript syntax, consider this function definition:

```
function square(x) { return x*x; }
```

This definition creates a new function object and assigns it to the variable `square`. The name of a function is really immaterial -- it is simply the name of a variable that holds the function. The function can be assigned to another variable and still work the same way:

```
var a = square(4); // a contains the number 16
var b = square;   // Now b refers to the same function that square
does
var c = b(5);     // c contains the number 25
```

Functions can also be assigned to object properties rather than global variables. When we do this, we call them methods:

```
var o = new Object;
o.square = new Function("x", "return x*x"); // Note Function( )
constructor
y = o.square(16);                          // y equals 256
```

Functions don't even require names at all, as when we assign them to array elements:

```
var a = new Array(3);
a[0] = function(x) { return x*x; } // Note function literal
a[1] = 20;
a[2] = a[0](a[1]);                 // a[2] contains 400
```

The function invocation syntax in this last example looks strange, but it is still a legal use of the JavaScript `()` operator!

[Example 7-2](#) is a detailed example of the things that can be done when functions are used as data. It demonstrates how functions can be passed as arguments to other functions and also how they can be stored in associative arrays (which were introduced in [Chapter 3](#) and are explained in detail in [Chapter 8](#).) This example may be a little tricky, but the comments explain what is going on; it is worth studying carefully.

Example 7-2. Using functions as data

```
// We define some simple functions here
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Here's a function that takes one of the above functions
```

```

// as an argument and invokes it on two operands
function operate(operator, operand1, operand2)
{
    return operator(operand1, operand2);
}

// We could invoke this function like this to compute the value (2+3) +
(4*5):
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// For the sake of example, we implement the functions again, this time
// using function literals. We store the functions in an associative
array.
var operators = new Object( );
operators["add"] = function(x,y) { return x+y; };
operators["subtract"] = function(x,y) { return x-y; };
operators["multiply"] = function(x,y) { return x*y; };
operators["divide"] = function(x,y) { return x/y; };
operators["pow"] = Math.pow; // Works for predefined functions too

// This function takes the name of an operator, looks up that operator
// in the array, and then invokes it on the supplied operands. Note
// the syntax used to invoke the operator function.
function operate2(op_name, operand1, operand2)
{
    if (operators[op_name] == null) return "unknown operator";
    else return operators[op_name](operand1, operand2);
}

// We could invoke this function as follows to compute
// the value ("hello" + " " + "world"):
var j = operate2("add", "hello", operate2("add", " ", "world"));
// Using the predefined Math.pow( ) function:
var k = operate2("pow", 10, 2)

```

If the preceding example does not convince you of the utility of being able to pass functions as arguments to other functions and otherwise treat functions as data values, consider the `Array.sort()` function. This function sorts the elements of an array. Because there are many possible orders to sort by (numerical order, alphabetical order, date order, ascending, descending, and so on), the `sort()` function optionally takes another function as an argument to tell it how to perform the sort. This function has a simple job -- it takes two elements of the array, compares them, and then returns a value that specifies which element comes first. This function argument makes the `Array.sort()` method perfectly general and infinitely flexible -- it can sort any type of data into any conceivable order!

7.3 Function Scope: The Call Object

As described in [Chapter 4](#), the body of a JavaScript function executes in a local scope that differs from the global scope. This new scope is created by adding the call object to the front of the scope chain. Since the call object is part of the scope chain, any properties

of this object are accessible as variables within the body of the function. Local variables declared with the `var` statement are created as properties of this object; the parameters of the function are also made available as properties of the object.

In addition to local variables and parameters, the call object defines one special property named `arguments`. This property refers to another special object known as the Arguments object, which is discussed in the next section. Because the `arguments` property is a property of the call object, it has exactly the same status as local variables and function parameters. For this reason, the identifier `arguments` should be considered a reserved word and should not be used as a variable or parameter name.

7.4 Function Arguments: The Arguments Object

Within the body of a function, the identifier `arguments` always has special meaning. `arguments` is a special property of the call object that refers to an object known as the *Arguments object*. The Arguments object is like an array that allows the argument values passed to the function to be retrieved by number, but it is not actually an Array object. The Arguments object also defines an additional `callee` property, described later.

Although a JavaScript function is defined with a fixed number of named arguments, it can be passed any number of arguments when it is invoked. The `arguments[]` array allows full access to these argument values, even when some are unnamed. Suppose you define a function `f` that expects to be passed one argument, `x`. If you invoke this function with two arguments, the first argument is accessible within the function by the parameter name `x` or as `arguments[0]`. The second argument is accessible only as `arguments[1]`. Furthermore, like all arrays, `arguments` has a `length` property that specifies the number of elements it contains. Thus, within the body of our function `f`, invoked with two arguments, `arguments.length` has the value 2.

The `arguments[]` array is useful in a number of ways. The following example shows how you can use it to check that a function is invoked with the correct number of arguments, since JavaScript doesn't do this for you:

```
function f(x, y, z)
{
    // First, check that the right number of arguments were passed
    if (arguments.length != 3) {
        throw new Error("function f called with " + arguments.length +
            "arguments, but it expects 3 arguments.");
    }
    // Now do the actual function...
}
```

The `arguments[]` array also opens up an important possibility for JavaScript functions: they can be written so that they work with any number of arguments. Here's an example that shows how you can write a simple `max()` function that accepts any number of

arguments and returns the value of the largest argument it is passed (see also the built-in function `Math.max()`, which in ECMAScript v3 also accepts any number of arguments):

```
function max( )
{
    var m = Number.NEGATIVE_INFINITY;

    // Loop through all the arguments, looking for, and
    // remembering, the biggest
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > m) m = arguments[i];
    // Return the biggest
    return m;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);
```

You can also use the `arguments[]` array to write functions that expect a fixed number of named arguments followed by an arbitrary number of unnamed arguments.

Throughout this section we've been referring to the "`arguments` array." Keep in mind, however, that `arguments` is not really an array; it is an Arguments object. Each Arguments object defines numbered array elements and a `length` property, but it is not technically an array -- it is better to think of it as an object that happens to have some numbered properties. The ECMAScript specification does not require the Arguments object to implement any of the special behavior that arrays do. Although you can assign a value to the `arguments.length` property, for example, ECMAScript does not require you to do so to actually alter the number of array elements defined in the object. (See [Chapter 9](#) for an explanation of the special behavior of the `length` property of true Array objects.)

The Arguments object has one *very* unusual feature. When a function has named arguments, the array elements of the Arguments object are synonyms for the local variables that hold the function arguments. The `arguments[]` array and the argument named `arguments` are two different ways of referring to the same variable. Changing the value of an argument with an argument name changes the value that is retrieved through the `arguments[]` array. Changing the value of an argument through the `arguments[]` array changes the value that is retrieved by the argument name. For example:

```
function f(x) {
    alert(x);           // Displays the initial value of the argument
    arguments[0] = null; // Changing the array element also changes x
    alert(x);           // Now displays "null"
}
```

7.4.1 The callee Property

In addition to its array elements, the Arguments object defines a `callee` property that refers to the function that is currently being executed. This is useful, for example, to allow unnamed functions to invoke themselves recursively. For instance, here is an unnamed function literal that computes factorials:

```
function(x) {
    if (x <= 1) return 1;
    return x * arguments.callee(x-1);
}
```

The `callee` property is defined by ECMAScript v1 and implemented in JavaScript 1.2.

7.5 Function Properties and Methods

We've seen that functions can be used as data values in JavaScript programs and that they can be created with the `Function()` constructor. These are sure signs that functions are actually represented by a type of JavaScript object, the Function object. Since functions are objects, they have properties and methods, just like the String and Date objects, for example. Now that we've discussed the call and Arguments objects that are used in the context of function invocation, let's turn to the Function object itself.

7.5.1 The length Property

As we've seen, within the body of a function, the `length` property of the `arguments` array specifies the number of arguments that were passed to the function. The `length` property of a function itself, however, has a different meaning. This read-only property returns the number of arguments that the function *expects* to be passed -- that is, the number of parameters it declares in its parameter list. Recall that a function can be invoked with any number of arguments, which it can retrieve through the `arguments` array, regardless of the number of parameters it declares. The `length` property of the Function object specifies exactly how many declared parameters a function has. Note that unlike `arguments.length`, this `length` property is available both inside and outside of the function body.

The following code defines a function named `check()` that is passed the `arguments` array from another function. It compares the `arguments.length` property to the `Function.length` property (which it accesses as `arguments.callee.length`) to see if the function was passed the number of arguments it expected. If not, it throws an exception. The `check()` function is followed by a test function `f()` that demonstrates how `check()` can be used:

```
function check(args) {
```

```

    var actual = args.length;           // The actual number of
arguments
    var expected = args.callee.length; // The expected number of
arguments
    if (actual != expected) { // Throw an exception if they don't
match
        throw new Error("Wrong number of arguments: expected: " +
            expected + "; actually passed " + actual);
    }
}

function f(x, y, z) {
    // Check that the actual # of args matches the expected # of args
    // Throw an exception if they don't match
    check(arguments);
    // Now do the rest of the function normally
    return x + y + z;
}

```

The `length` property of the Function object is standardized by ECMAScript v1 and implemented in JavaScript 1.1 and later.^[3]

^[3] In Netscape 4.0, a bug prevents this property from working correctly unless the `language` attribute of the `<script>` tag is explicitly set to "JavaScript1.2".

7.5.2 The prototype Property

Every function has a `prototype` property that refers to a predefined *prototype object*. This prototype object comes into play when the function is used as a constructor with the `new` operator; it plays an important role in the process of defining new object types. We'll explore this property in detail in [Chapter 8](#).

7.5.3 Defining Your Own Function Properties

When a function needs to use a variable whose value persists across invocations, it is often convenient to use a property of the Function object, instead of cluttering up the namespace by defining a global variable. For example, suppose we want to write a function that returns a unique identifier whenever it is invoked. The function must never return the same value twice. In order to manage this, the function needs to keep track of the values it has already returned, and this information must persist across function invocations. We could store this information in a global variable, but that is unnecessary because the information is used only by the function itself. It is better to store the information in a property of the Function object. Here is an example that returns a unique integer whenever it is called:

```

// Create and initialize the "static" variable.
// Function declarations are processed before code is executed, so
// we really can do this assignment before the function declaration.
uniqueInteger.counter = 0;

```

```
// Here's the function. It returns a different value each time
// it is called and uses a "static" property of itself to keep track
// of the last value it returned.
function uniqueInteger( ) {
    // Increment and return our "static" variable
    return uniqueInteger.counter++;
}
```

7.5.4 The `apply()` and `call()` Methods

ECMAScript v3 defines two methods that are defined for all functions, `call()` and `apply()`. These methods allow you to invoke a function as if it were a method of some other object. (Note that we have not discussed methods yet; you may find this section more understandable once you have read [Chapter 8](#).) The first argument to both `call()` and `apply()` is the object on which the function is to be invoked; this argument becomes the value of the `this` keyword within the body of the function. Any remaining arguments to `call()` are the values that are passed to the function that is invoked. For example, to pass two numbers to the function `f()` and invoke it as if it were a method of the object `o`, you could use code like this:

```
f.call(o, 1, 2);
```

This is similar to the following lines of code:

```
o.m = f;
o.m(1,2);
delete o.m;
```

The `apply()` method is like the `call()` method, except that the arguments to be passed to the function are specified as an array:

```
f.apply(o, [1,2]);
```

For example, to find the largest number in an array of numbers, you could use the `apply()` method to pass the elements of the array to the `Math.max()` function:^[4]

^[4] This example assumes we are using the ECMAScript v3 `Math.max()` function, which accepts an arbitrary number of arguments; the ECMAScript v1 version of the function accepts only two arguments.

```
var biggest = Math.max.apply(null, array_of_numbers);
```

The `apply()` method is implemented in JavaScript 1.2, but the `call()` method is not implemented until JavaScript 1.5

Chapter 8. Objects

[Chapter 3](#) explained that objects are one of the fundamental data types in JavaScript. They are also one of the most important. This chapter describes JavaScript objects in detail. Basic usage of objects, described in the next section, is straightforward, but as we'll see in later sections, objects have more complex uses and behaviors.

8.1 Objects and Properties

Objects are composite data types: they aggregate multiple values into a single unit and allow us to store and retrieve those values by name. Another way to explain this is to say that an object is an unordered collection of *properties*, each of which has a name and a value. The named values held by an object may be primitive values like numbers and strings, or they may themselves be objects.

8.1.1 Creating Objects

Objects are created with the `new` operator. This operator must be followed by the name of a constructor function that serves to initialize the object. For example, we can create an empty object (an object with no properties) like this:

```
var o = new Object( );
```

JavaScript supports other built-in constructor functions that initialize newly created objects in other, less trivial, ways. For example, the `Date()` constructor initializes an object that represents a date and time:

```
var now = new Date( ); // The current date and
time
var new_years_eve = new Date(2000, 11, 31); // Represents December 31,
2000
```

Later in this chapter, we'll see that it is possible to define custom constructor methods to initialize newly created objects in any way you desire.

Object literals provide another way to create and initialize new objects. As we saw in [Chapter 3](#), an object literal allows us to embed an object description literally in JavaScript code in much the same way that we embed textual data into JavaScript code as quoted strings. An object literal consists of a comma-separated list of property specifications enclosed within curly braces. Each property specification in an object literal consists of the property name followed by a colon and the property value. For example:

```
var circle = { x:0, y:0, radius:2 }
var homer = {
```



```
        name: "Homer Simpson",
        age: 34,
        married: true,
        occupation: "plant operator",
        email: "homer@simpsons.com"
    };
```

The object literal syntax is defined by the ECMAScript v3 specification and implemented in JavaScript 1.2 and later.

8.1.2 Setting and Querying Properties

You normally use the `.` operator to access the value of an object's properties. The value on the left of the `.` should be a reference to an object (usually just the name of the variable that contains the object reference). The value on the right of the `.` should be the name of the property. This must be an identifier, not a string or an expression. For example, you would refer to the property `p` in object `o` with `o.p` or to the property `radius` in the object `circle` with `circle.radius`. Object properties work like variables: you can store values in them and read values from them. For example:

```
// Create an object. Store a reference to it in a variable.
var book = new Object( );

// Set a property in the object.
book.title = "JavaScript: The Definitive Guide"

// Set some more properties. Note the nested objects.
book.chapter1 = new Object( );
book.chapter1.title = "Introduction to JavaScript";
book.chapter1.pages = 19;
book.chapter2 = { title: "Lexical Structure", pages: 6 };

// Read some property values from the object.
alert("Outline: " + book.title + "\n\t" +
      "Chapter 1 " + book.chapter1.title + "\n\t" +
      "Chapter 2 " + book.chapter2.title);
```

An important point to notice about this example is that you can create a new property of an object simply by assigning a value to it. Although we declare variables with the `var` keyword, there is no need (and no way) to do so with object properties. Furthermore, once you have created an object property by assigning a value to it, you can change the value of the property at any time simply by assigning a new value:

```
book.title = "JavaScript: The Rhino Book"
```

8.1.3 Enumerating Properties

The `for/in` loop discussed in [Chapter 6](#) provides a way to loop through, or enumerate, the properties of an object. This can be useful when debugging scripts or when working with objects that may have arbitrary properties whose names you do not know in advance. The following code shows a function you can use to list the property names of an object:

```
function DisplayPropertyNames(obj) {
    var names = "";
    for(var name in obj) names += name + "\n";
    alert(names);
}
```

Note that the `for/in` loop does not enumerate properties in any specific order, and although it enumerates all user-defined properties, it does not enumerate certain predefined properties or methods.

8.1.4 Undefined Properties

If you attempt to read the value of a property that does not exist (in other words, a property that has never had a value assigned to it), you end up retrieving the `undefined` value (introduced in [Chapter 3](#)).

You can use the `delete` operator to delete a property of an object:

```
delete book.chapter2;
```

Note that deleting a property does not merely set the property to `undefined`; it actually removes the property from the object. The `for/in` loop demonstrates this difference: it enumerates properties that have been set to the `undefined` value, but it does not enumerate deleted properties.

8.2 Constructors

We saw previously that you can create and initialize a new object in JavaScript by using the `new` operator in conjunction with a predefined constructor function such as `Object()`, `Date()`, or `Function()`. These predefined constructors and the built-in object types they create are useful in many instances. However, in object-oriented programming, it is also common to work with custom object types defined by your program. For example, if you are writing a program that manipulates rectangles, you might want to represent rectangles with a special type, or *class*, of object. Each object of this `Rectangle` class would have a `width` property and a `height` property, since those are the essential defining characteristics of rectangles.

To create objects with properties such as `width` and `height` already defined, we need to write a constructor to create and initialize these properties in a new object. A constructor is a JavaScript function with two special features:

- It is invoked through the `new` operator.
- It is passed a reference to a newly created, empty object as the value of the `this` keyword, and it is responsible for performing appropriate initialization for that new object.

[Example 8-1](#) shows how the constructor function for a `Rectangle` object might be defined and invoked.

Example 8-1. A Rectangle object constructor function

```
// Define the constructor.
// Note how it initializes the object referred to by "this".
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
}

// Invoke the constructor to create two Rectangle objects.
// We pass the width and height to the constructor,
// so it can initialize each new object appropriately.
var rect1 = new Rectangle(2, 4);
var rect2 = new Rectangle(8.5, 11);
```

Notice how the constructor uses its arguments to initialize properties of the object referred to by the `this` keyword. Keep in mind that a constructor function simply initializes the specified object; it does not have to return that object.

We have defined a class of objects simply by defining an appropriate constructor function -- all objects created with the `Rectangle()` constructor are now guaranteed to have initialized `width` and `height` properties. This means that we can write programs that rely on this fact and treat all `Rectangle` objects uniformly. Because every constructor defines a class of objects, it is stylistically important to give a constructor function a name that indicates the class of objects it creates. Creating a rectangle with `new Rectangle(1,2)` is a lot more intuitive than with `new init_rect(1,2)`, for example.

Constructor functions typically do not have return values. They initialize the object passed as the value of `this` and return nothing. However, a constructor is allowed to return an object value, and, if it does so, that returned object becomes the value of the `new` expression. In this case, the object that was the value of `this` is simply discarded.

8.3 Methods

A *method* is nothing more than a JavaScript function that is invoked through an object. Recall that functions are data values and that there is nothing special about the names with which they are defined -- a function can be assigned to any variable, or even to any property of an object. If we have a function `f` and an object `o`, we can define a method named `m` with the following line:

```
o.m = f;
```

Having defined the method `m()` of the object `o`, we invoke it like this:

```
o.m( );
```

Or, if `m()` expects two arguments, we might invoke it like this:

```
o.m(x, x+2);
```

Methods have one very important property: the object through which a method is invoked becomes the value of the `this` keyword within the body of the method. For example, when we invoke `o.m()`, the body of the method can refer to the object `o` with the `this` keyword.

The discussion of the `this` keyword should begin to clarify why we use methods at all. Any function that is used as a method is effectively passed an extra argument -- the object through which it is invoked. Typically, a method performs some sort of operation on that object, so the method invocation syntax is a particularly elegant way to express the fact that a function is operating on an object. Compare the following two lines of code:

```
rect.setSize(width, height);  
setRectSize(rect, width, height);
```

These two lines may perform exactly the same operation on the object `rect`, but the method invocation syntax in the first line more clearly indicates the idea that it is the object `rect` that is the primary focus, or *target*, of the operation. (If the first line does not seem a more natural syntax to you, you are probably new to object-oriented programming. With a little experience, you will learn to love it!)

While it is useful to think of functions and methods differently, there is not actually as much difference between them as there initially appears to be. Recall that functions are values stored in variables and that variables are nothing more than properties of a global object. Thus, when you invoke a function, you are actually invoking a method of the global object. Within such a function, the `this` keyword refers to the global object. Thus, there is no technical difference between functions and methods. The real difference lies in

design and intent: methods are written to operate somehow on the `this` object, while functions usually stand alone and do not use the `this` object.

The typical usage of methods is more clearly illustrated through an example. [Example 8-2](#) returns to the Rectangle objects of [Example 8-1](#) and shows how a method that operates on Rectangle objects can be defined and invoked.

Example 8-2. Defining and invoking a method

```
// This function uses the this keyword, so it doesn't make sense to
// invoke it by itself; it needs instead to be made a method of some
// object that has "width" and "height" properties defined.
function compute_area( )
{
    return this.width * this.height;
}

// Create a new Rectangle object, using the constructor defined
// earlier.
var page = new Rectangle(8.5, 11);

// Define a method by assigning the function to a property of the
// object.
page.area = compute_area;

// Invoke the new method like this:
var a = page.area( );    // a = 8.5*11 = 93.5
```

One shortcoming is evident in [Example 8-2](#): before you can invoke the `area()` method for the `rect` object, you must assign that method to a property of the object. While we can invoke the `area()` method on the particular object named `page`, we can't invoke it on any other Rectangle objects without first assigning the method to them. This quickly becomes tedious. [Example 8-3](#) defines some additional Rectangle methods and shows how they can automatically be assigned to all Rectangle objects with a constructor function.

Example 8-3. Defining methods in a constructor

```
// First, define some functions that will be used as methods.
function Rectangle_area( ) { return this.width * this.height; }
function Rectangle_perimeter( ) { return 2*this.width + 2*this.height; }
}
function Rectangle_set_size(w,h) { this.width = w; this.height = h; }
function Rectangle_enlarge( ) { this.width *= 2; this.height *= 2; }
function Rectangle_shrink( ) { this.width /= 2; this.height /= 2; }

// Then define a constructor method for our Rectangle objects.
// The constructor initializes properties and also assigns methods.
function Rectangle(w, h)
{
    // Initialize object properties.
    this.width = w;
    this.height = h;
```

```

    // Define methods for the object.
    this.area = Rectangle_area;
    this.perimeter = Rectangle_perimeter;
    this.set_size = Rectangle_set_size;
    this.enlarge = Rectangle_enlarge;
    this.shrink = Rectangle_shrink;
}
// Now, when we create a rectangle, we can immediately invoke methods
on it:
var r = new Rectangle(2,2);
var a = r.area( );
r.enlarge( );
var p = r.perimeter( );

```

The technique shown in [Example 8-3](#) also has a shortcoming. In this example, the `Rectangle()` constructor sets seven properties of each and every `Rectangle` object it initializes, even though five of those properties have constant values that are the same for every rectangle. Each property takes up memory space; by adding methods to our `Rectangle` class, we've more than tripled the memory requirements of each `Rectangle` object. Fortunately, JavaScript has a solution to this problem: it allows an object to inherit properties from a prototype object. The next section describes this technique in detail.

8.4 Prototypes and Inheritance

We've seen how inefficient it can be to use a constructor to assign methods to the objects it initializes. When we do this, each and every object created by the constructor has identical copies of the same method properties. There is a much more efficient way to specify methods, constants, and other properties that are shared by all objects in a class.

JavaScript objects "inherit" properties from a *prototype object*.^[1] Every object has a prototype; all of the properties of the prototype object appear to be properties of any objects for which it is a prototype. That is, each object *inherits* properties from its prototype.

^[1] Prototypes were introduced in JavaScript 1.1; they are not supported in the now obsolete JavaScript 1.0.

The prototype of an object is defined by the constructor function that was used to create and initialize the object. All functions in JavaScript have a `prototype` property that refers to an object. This prototype object is initially empty, but any properties you define in it will be inherited by all objects created by the constructor.

A constructor defines a class of objects and initializes properties, such as `width` and `height`, that are the state variables for the class. The prototype object is associated with the constructor, so each member of the class inherits exactly the same set of properties from the prototype. This means that the prototype object is an ideal place for methods and other constant properties.

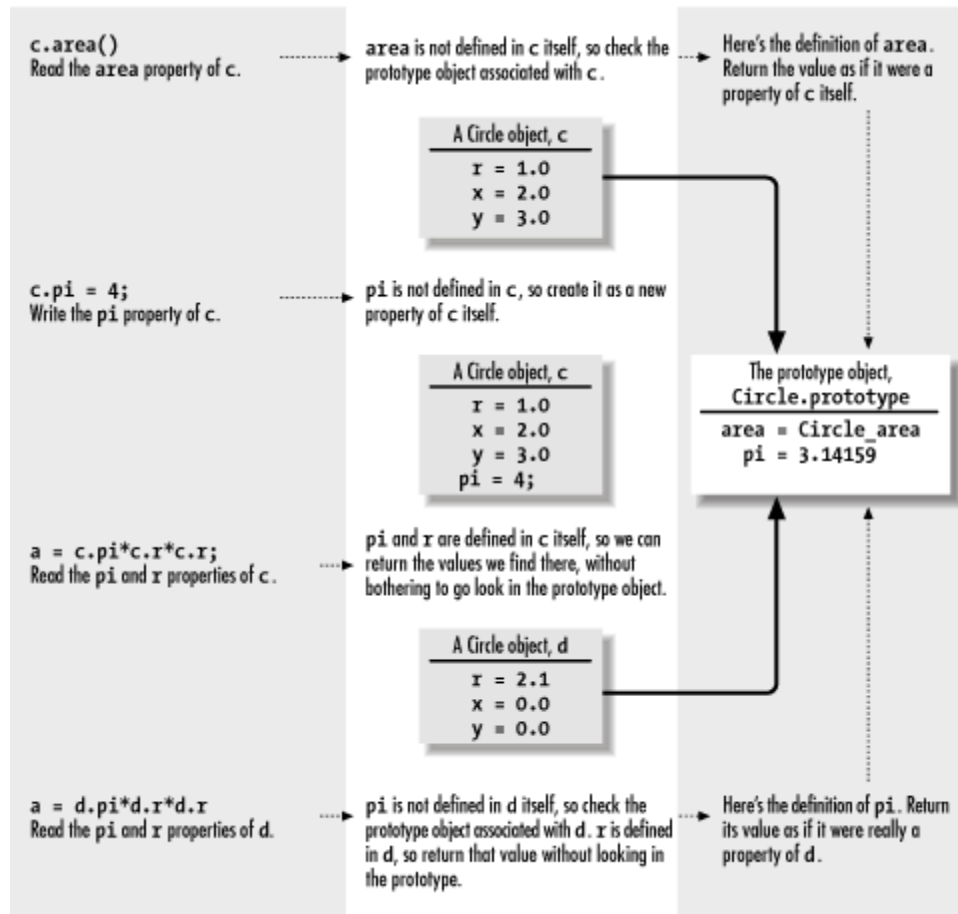
Note that inheritance occurs automatically, as part of the process of looking up a property value. Properties are *not* copied from the prototype object into new objects; they merely appear as if they were properties of those objects. This has two important implications. First, the use of prototype objects can dramatically decrease the amount of memory required by each object, since the object can inherit many of its properties. The second implication is that an object inherits properties even if they are added to its prototype *after* the object is created.

Each class has one prototype object, with one set of properties. But there are potentially many instances of a class, each of which inherits those prototype properties. Because one prototype property can be inherited by many objects, JavaScript must enforce a fundamental asymmetry between reading and writing property values. When you read property `p` of an object `o`, JavaScript first checks to see if `o` has a property named `p`. If it does not, it next checks to see if the prototype object of `o` has a property named `p`. This is what makes prototype-based inheritance work.

When you write the value of a property, on the other hand, JavaScript does not use the prototype object. To see why, consider what would happen if it did: suppose you try to set the value of the property `o.p` when the object `o` does not have a property named `p`. Further suppose that JavaScript goes ahead and looks up the property `p` in the prototype object of `o` and allows you to set the property of the prototype. Now you have changed the value of `p` for a whole class of objects -- not at all what you intended.

Therefore, property inheritance occurs only when you read property values, not when you write them. If you set the property `p` in an object `o` that inherits that property from its prototype, what happens is that you create a new property `p` directly in `o`. Now that `o` has its own property named `p`, it no longer inherits the value of `p` from its prototype. When you read the value of `p`, JavaScript first looks at the properties of `o`. Since it finds `p` defined in `o`, it doesn't need to search the prototype object and never finds the value of `p` defined there. We sometimes say that the property `p` in `o` "shadows" or "hides" the property `p` in the prototype object. Prototype inheritance can be a confusing topic. [Figure 8-1](#) illustrates the concepts we've discussed here.

Figure 8-1. Objects and prototypes



Because prototype properties are shared by all objects of a class, it generally makes sense to use them only to define properties that are the same for all objects within the class. This makes prototypes ideal for defining methods. Other properties with constant values (such as mathematical constants) are also suitable for definition with prototype properties. If your class defines a property with a very commonly used default value, you might define this property and its default value in a prototype object. Then, the few objects that want to deviate from the default value can create their own private, unshared copies of the property and define their own nondefault values.

Let's move from an abstract discussion of prototype inheritance to a concrete example. Suppose we define a `Circle()` constructor function to create objects that represent circles. The prototype object for this class is `Circle.prototype`,^[2] so we can define a constant available to all `Circle` objects like this:

^[2] The prototype object of a constructor is created automatically by JavaScript. In most versions of JavaScript, every function is automatically given an empty prototype object, just in case it is used as a constructor. In JavaScript 1.1, however, the prototype object is not created until the function is used as a constructor for the first time. This means that if you require compatibility with JavaScript 1.1, you should create at least one object of a class before you use the prototype object to assign methods and constants to objects of that class. So, if we have defined a `Circle()` constructor but have not yet used it to create any `Circle` objects, we'd define the constant property `pi` like this:


```
//First create and discard a dummy object; forces prototype object
creation. new Circle ( ); //Now we can set properties in the prototype.
Circle.prototype.pi = 3.14159'
```

```
Circle.prototype.pi = 3.14159;
```

[Example 8-4](#) shows our Circle example fully fleshed out. The code defines a `Circle` class by first defining a `Circle()` constructor to initialize each individual object and then setting properties of `Circle.prototype` to define methods and constants shared by all instances of the class.

Example 8-4. Defining a Circle class with a prototype object

```
// Define a constructor method for our class.
// Use it to initialize properties that will be different for
// each individual Circle object.
function Circle(x, y, r)
{
    this.x = x; // The X-coordinate of the center of the circle
    this.y = y; // The Y-coordinate of the center of the circle
    this.r = r; // The radius of the circle
}

// Create and discard an initial Circle object.
// This forces the prototype object to be created in JavaScript 1.1.
new Circle(0,0,0);

// Define a constant: a property that will be shared by
// all circle objects. Actually, we could just use Math.PI,
// but we do it this way for the sake of instruction.
Circle.prototype.pi = 3.14159;

// Define a method to compute the circumference of the circle.
// First declare a function, then assign it to a prototype property.
// Note the use of the constant defined above.
function Circle_circumference( ) { return 2 * this.pi * this.r; }
Circle.prototype.circumference = Circle_circumference;

// Define another method. This time we use a function literal to define
// the function and assign it to a prototype property all in one step.
Circle.prototype.area = function( ) { return this.pi * this.r *
this.r; }

// The Circle class is defined.
// Now we can create an instance and invoke its methods.
var c = new Circle(0.0, 0.0, 1.0);
var a = c.area( );
var p = c.circumference( );
```

8.4.1 Prototypes and Built-in Classes

It is not only user-defined classes that have prototype objects. Built-in classes, such as `String` and `Date`, have prototype objects too, and you can assign values to them.^[43] For example, the following code defines a new method that is available for all `String` objects:

^[43] In JavaScript 1.1 and later.

```
// Returns true if the last character is c
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}
```

Having defined the new `endsWith()` method in the `String` prototype object, we can use it like this:

```
var message = "hello world";
message.endsWith('h') // Returns false
message.endsWith('d') // Returns true
```

8.5 Object-Oriented JavaScript

Although JavaScript supports a data type we call an object, it does not have a formal notion of a class. This makes it quite different from classic object-oriented languages such as C++ and Java. The common conception about object-oriented programming languages is that they are strongly typed and support class-based inheritance. By these criteria, it is easy to dismiss JavaScript as not being a true object-oriented language. On the other hand, we've seen that JavaScript makes heavy use of objects and that it has its own type of prototype-based inheritance. JavaScript is a true object-oriented language. It draws inspiration from a number of other (relatively obscure) object-oriented languages that use prototype-based inheritance instead of class-based inheritance.

Although JavaScript is not a class-based object-oriented language, it does a good job of simulating the features of class-based languages such as Java and C++. I've been using the term class informally throughout this chapter. This section more formally explores the parallels between JavaScript and true class-based inheritance languages such as Java and C++.^[44]

^[44] You should read this section even if you are not familiar with those languages and that style of object-oriented programming.

Let's start by defining some basic terminology. An *object*, as we've already seen, is a data structure that contains various pieces of named data and may also contain various methods to operate on those pieces of data. An object groups related values and methods into a single convenient package, which generally makes programming easier by increasing the modularity and reusability of code. Objects in JavaScript may have any number of properties, and properties may be dynamically added to an object. This is not the case in strictly typed languages such as Java and C++. In those languages, each object

has a predefined set of properties,^[5] where each property is of a predefined type. When we are using JavaScript objects to simulate object-oriented programming techniques, we generally define in advance the set of properties for each object and the type of data that each property holds.

^[5] They are usually called "fields" in Java and C++, but we'll refer to them as properties here, since that is the JavaScript terminology.

In Java and C++, a *class* defines the structure of an object. The class specifies exactly what fields an object contains and what types of data each holds. It also defines the methods that operate on an object. JavaScript does not have a formal notion of a class, but, as we've seen, it approximates classes with its constructors and their prototype objects.

In both JavaScript and class-based object-oriented languages, there may be multiple objects of the same class. We often say that an object is an *instance* of its class. Thus, there may be many instances of any class. Sometimes we use the term *instantiate* to describe the process of creating an object (i.e., an instance of a class).

In Java, it is a common programming convention to name classes with an initial capital letter and to name objects with lowercase letters. This convention helps keep classes and objects distinct from each other in code; it is a useful convention to follow in JavaScript programming as well. In previous sections, for example, we've defined the Circle and Rectangle classes and have created instances of those classes named `c` and `rect`.

The members of a Java class may be of four basic types: instance properties, instance methods, class properties, and class methods. In the following sections, we'll explore the differences between these types and show how they are simulated in JavaScript.

8.5.1 Instance Properties

Every object has its own separate copies of its *instance properties*. In other words, if there are 10 objects of a given class, there are 10 copies of each instance property. In our Circle class, for example, every Circle object has a property `r` that specifies the radius of the circle. In this case, `r` is an instance property. Since each object has its own copy of the instance properties, these properties are accessed through individual objects. If `c` is an object that is an instance of the Circle class, for example, we refer to its radius as:

```
c.r
```

By default, any object property in JavaScript is an instance property. To truly simulate object-oriented programming, however, we will say that instance properties in JavaScript are those properties that are created and/or initialized in an object by the constructor function.

8.5.2 Instance Methods

An *instance method* is much like an instance property, except that it is a method rather than a data value. (In Java, functions and methods are not data, as they are in JavaScript, so this distinction is more clear.) Instance methods are invoked on a particular object, or instance. The `area()` method of our `Circle` class is an instance method. It is invoked on a `Circle` object `c` like this:

```
a = c.area( );
```

Instance methods use the `this` keyword to refer to the object or instance on which they are operating. An instance method can be invoked for any instance of a class, but this does not mean that each object contains its own private copy of the method, as it does with instance properties. Instead, each instance method is shared by all instances of a class. In JavaScript, we define an instance method for a class by setting a property in the constructor's prototype object to a function value. This way, all objects created by that constructor share an inherited reference to the function and can invoke it using the method invocation syntax shown earlier.

8.5.3 Class Properties

A *class property* in Java is a property that is associated with a class itself, rather than with each instance of a class. No matter how many instances of the class are created, there is only one copy of each class property. Just as instance properties are accessed through an instance of a class, class properties are accessed through the class itself.

`Number.MAX_VALUE` is an example of a class property in JavaScript: the `MAX_VALUE` property is accessed through the `Number` class. Because there is only one copy of each class property, class properties are essentially global. What is nice about them, however, is that they are associated with a class and they have a logical niche, a position in the JavaScript namespace, where they are not likely to be overwritten by other properties with the same name. As is probably clear, we simulate a class property in JavaScript simply by defining a property of the constructor function itself. For example, to create a class property `Circle.PI` to store the mathematical constant π , we can do the following:

```
Circle.PI = 3.14;
```

`Circle` is a constructor function, but because JavaScript functions are objects, we can create properties of a function just as we can create properties of any other object.

8.5.4 Class Methods

Finally, we come to class methods. A *class method* is a method associated with a class rather than with an instance of a class; they are invoked through the class itself, not through a particular instance of the class. The `Date.parse()` method (which you can

look up in the core reference section of this book) is a class method. You always invoke it through the Date constructor object, rather than through a particular instance of the Date class.

Because class methods are not invoked through a particular object, they cannot meaningfully use the `this` keyword -- `this` refers to the object for which an instance method is invoked. Like class properties, class methods are global. Because they do not operate on a particular object, class methods are generally more easily thought of as functions that happen to be invoked through a class. Again, associating these functions with a class gives them a convenient niche in the JavaScript namespace and prevents namespace collisions. To define a class method in JavaScript, we simply make the appropriate function a property of the constructor.

8.5.5 Example: The Circle Class

[Example 8-5](#) is a reimplementaion of our Circle class that contains examples of each of these four basic types of members.

Example 8-5. Defining instance and class properties and methods

```
function Circle(radius) { // The constructor defines the class
  itself.
  // r is an instance property, defined and initialized in the
  constructor.
  this.r = radius;
}

// Circle.PI is a class property--it is a property of the constructor
function.
Circle.PI = 3.14159;

// Here is a function that computes a circle's area.
function Circle_area( ) { return Circle.PI * this.r * this.r; }

// Here we make the function into an instance method by assigning it
// to the prototype object of the constructor.
// Note: with JavaScript 1.2, we can use a function literal to
// define the function without naming it Circle_area.
Circle.prototype.area = Circle_area;

// Here's another function. It takes two Circle objects as arguments
and
// returns the one that is larger (i.e., has the larger radius).
function Circle_max(a,b) {
  if (a.r > b.r) return a;
  else return b;
}

// Since this function compares two Circle objects, it doesn't make
sense as
// an instance method operating on a single Circle object. But we don't
want
```

```

// it to be a standalone function either, so we make it into a class
method
// by assigning it to the constructor function:
Circle.max = Circle_max;

// Here is some code that uses each of these fields:
var c = new Circle(1.0);      // Create an instance of the Circle class
c.r = 2.2;                   // Set the r instance property
var a = c.area();            // Invoke the area( ) instance method
var x = Math.exp(Circle.PI); // Use the PI class property in our own
computation
var d = new Circle(1.2);      // Create another Circle instance
var bigger = Circle.max(c,d); // Use the max( ) class method

```

8.5.6 Example: Complex Numbers

[Example 8-6](#) is another example, somewhat more formal than the last, of defining a class of objects in JavaScript. The code and the comments are worth careful study. Note that this example uses the function literal syntax of JavaScript 1.2. Because it requires this version of the language (or later), it does not bother with the JavaScript 1.1 compatibility technique of invoking the constructor once before assigning to its prototype object.

Example 8-6. A complex number class

```

/*
 * Complex.js:
 * This file defines a Complex class to represent complex numbers.
 * Recall that a complex number is the sum of a real number and an
 * imaginary number and that the imaginary number i is the
 * square root of -1.
 */

/*
 * The first step in defining a class is defining the constructor
 * function of the class. This constructor should initialize any
 * instance properties of the object. These are the essential
 * "state variables" that make each instance of the class different.
 */
function Complex(real, imaginary) {
    this.x = real;      // The real part of the number
    this.y = imaginary; // The imaginary part of the number
}

/*
 * The second step in defining a class is defining its instance
 * methods (and possibly other properties) in the prototype object
 * of the constructor. Any properties defined in this object will
 * be inherited by all instances of the class. Note that instance
 * methods operate implicitly on the this keyword. For many methods,
 * no other arguments are needed.
 */

// Return the magnitude of a complex number. This is defined
// as its distance from the origin (0,0) of the complex plane.

```

```

Complex.prototype.magnitude = function( ) {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};

// Return a complex number that is the negative of this one.
Complex.prototype.negative = function( ) {
    return new Complex(-this.x, -this.y);
};

// Convert a Complex object to a string in a useful way.
// This is invoked when a Complex object is used as a string.
Complex.prototype.toString = function( ) {
    return "{" + this.x + "," + this.y + "}";
};

// Return the real portion of a complex number. This function
// is invoked when a Complex object is treated as a primitive value.
Complex.prototype.valueOf = function( ) { return this.x; }

/*
 * The third step in defining a class is to define class methods,
 * constants, and any needed class properties as properties of the
 * constructor function itself (instead of as properties of the
 * prototype object of the constructor). Note that class methods
 * do not use the this keyword: they operate only on their arguments.
 */

// Add two complex numbers and return the result.
Complex.add = function (a, b) {
    return new Complex(a.x + b.x, a.y + b.y);
};

// Subtract one complex number from another.
Complex.subtract = function (a, b) {
    return new Complex(a.x - b.x, a.y - b.y);
};

// Multiply two complex numbers and return the product.
Complex.multiply = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
        a.x * b.y + a.y * b.x);
};

// Here are some useful predefined complex numbers.
// They are defined as class properties, where they can be used as
// "constants." (Note, though, that they are not actually read-only.)
Complex.zero = new Complex(0,0);
Complex.one = new Complex(1,0);
Complex.i = new Complex(0,1);

```

8.5.7 Superclasses and Subclasses

In Java, C++, and other class-based object-oriented languages, there is an explicit concept of the *class hierarchy*. Every class can have a *superclass* from which it inherits properties

and methods. Any class can be extended, or subclassed, so that the resulting *subclass* inherits its behavior. As we've seen, JavaScript supports prototype inheritance instead of class-based inheritance. Still, JavaScript analogies to the class hierarchy can be drawn. In JavaScript, the `Object` class is the most generic, and all other classes are specialized versions, or subclasses, of it. Another way to say this is that `Object` is the superclass of all the built-in classes. All classes inherit a few basic methods (described later in this chapter) from `Object`.

We've learned that objects inherit properties from the prototype object of their constructor. How do they also inherit properties from the `Object` class? Remember that the prototype object is itself an object; it is created with the `Object()` constructor. This means the prototype object itself inherits properties from `Object.prototype`! So, an object of class `Complex` inherits properties from the `Complex.prototype` object, which itself inherits properties from `Object.prototype`. Thus, the `Complex` object inherits properties of both objects. When you look up a property in a `Complex` object, the object itself is searched first. If the property is not found, the `Complex.prototype` object is searched next. Finally, if the property is not found in that object, the `Object.prototype` object is searched.

Note that because the `Complex` prototype object is searched before the `Object` prototype object, properties of `Complex.prototype` hide any properties with the same name in `Object.prototype`. For example, in the class definition shown in [Example 8-6](#), we defined a `toString()` method in the `Complex.prototype` object. `Object.prototype` also defines a method with this name, but `Complex` objects never see it because the definition of `toString()` in `Complex.prototype` is found first.

The classes we've shown in this chapter are all direct subclasses of `Object`. This is typical of JavaScript programming; there is not usually any need to produce a more complex class hierarchy. When necessary, however, it is possible to subclass any other class. For example, suppose we want to produce a subclass of `Complex` in order to add some more methods. To do this, we simply have to make sure that the prototype object of the new class is itself an instance of `Complex`, so that it inherits all the properties of `Complex.prototype`:

```
// This is the constructor for the subclass.
function MoreComplex(real, imaginary) {
    this.x = real;
    this.y = imaginary;
}

// We force its prototype to be a Complex object. This means that
// instances of our new class inherit from MoreComplex.prototype,
// which inherits from Complex.prototype, which inherits from
// Object.prototype.
MoreComplex.prototype = new Complex(0,0);

// Now add a new method or other new features to this subclass.
MoreComplex.prototype.swap = function( ) {
    var tmp = this.x;
```



```
    this.x = this.y;
    this.y = tmp;
}
```

There is one subtle shortcoming to the subclassing technique shown here. Since we explicitly set `MoreComplex.prototype` to an object of our own creation, we overwrite the prototype object provided by JavaScript and discard the `constructor` property we are given. This `constructor` property, described later in this chapter, is supposed to refer to the constructor function that created the object. A `MoreComplex` object, however, inherits the `constructor` property of its superclass, rather than having one of its own. One solution is to set this property explicitly:

```
MoreComplex.prototype.constructor = MoreComplex;
```

Note, however, that in JavaScript 1.1, the `constructor` property is read-only and cannot be set in this way.

8.6 Objects as Associative Arrays

We've seen the `.` operator used to access the properties of an object. It is also possible to use the `[]` operator, which is more commonly used with arrays, to access these properties. Thus, the following two JavaScript expressions have the same value:

```
object.property
object["property"]
```

The important difference to note between these two syntaxes is that in the first, the property name is an identifier, and in the second, the property name is a string. We'll see why this is so important shortly.

In C, C++, Java, and similar strongly typed languages, an object can have only a fixed number of properties, and the names of these properties must be defined in advance. Since JavaScript is a loosely typed language, this rule does not apply -- a program can create any number of properties in any object. When you use the `.` operator to access a property of an object, however, the name of the property is expressed as an identifier. Identifiers must be typed literally into your JavaScript program -- they are not a data type, so they cannot be manipulated by the program.

On the other hand, when you access a property of an object with the `[]` array notation, the name of the property is expressed as a string. Strings are JavaScript data types, so they can be manipulated and created while a program is running. So, for example, you could write the following code in JavaScript:

```
var addr = "";
```

```
for(i = 0; i < 4; i++) {  
    addr += customer["address" + i] + '\n';  
}
```

This code reads and concatenates the `address0`, `address1`, `address2`, and `address3` properties of the `customer` object.

This brief example demonstrates the flexibility of using array notation to access properties of an object with string expressions. We could have written this example using the `.` notation, but there are cases where only the array notation will do. Suppose, for example, that you are writing a program that uses network resources to compute the current value of the user's stock market investments. The program allows the user to type in the name of each stock she owns, as well as the number of shares of each stock. You might use an object named `portfolio` to hold this information. The object has one property for each stock. The name of the property is the name of the stock and the property value is the number of shares of that stock. So, for example, if a user holds 50 shares of stock in IBM, the `portfolio.ibm` property has the value 50.

Part of this program needs to have a loop that prompts the user to enter the name of a stock she owns and then asks her to enter the number of shares she owns of that stock. Inside the loop, you'd have code something like this:

```
var stock_name = get_stock_name_from_user( );  
var shares = get_number_of_shares( );  
portfolio[stock_name] = shares;
```

Since the user enters stock names at runtime, there is no way that you can know the property names ahead of time. Since you can't know the property names when you write the program, there is no way you can use the `.` operator to access the properties of the `portfolio` object. You can use the `[]` operator, however, because it uses a string value (which is dynamic and can change at runtime), rather than an identifier (which is static and must be hardcoded in the program), to name the property.

When an object is used this fashion, it is often called an *associative array* -- a data structure that allows you to dynamically associate arbitrary values with arbitrary strings. JavaScript objects are actually implemented internally as associative arrays. The `.` notation for accessing properties makes them seem like the static objects of C++ and Java, and they work perfectly well in that capacity. But they also have the powerful ability to associate values with arbitrary strings. In this respect, JavaScript objects are much more like Perl arrays than C++ or Java objects.

[Chapter 6](#) introduced the `for/in` loop. The real power of this JavaScript statement becomes clear when we consider its use with associative arrays. To return to the stock portfolio example, we might use the following code after the user has entered her portfolio and we are computing its current total value:

```
var value = 0;
for (stock in portfolio) {
    // For each stock in the portfolio, get the per share value
    // and multiply it by the number of shares.
    value += get_share_value(stock) * portfolio[stock];
}
```

We cannot write this code without the `for/in` loop because the names of the stocks aren't known in advance. This is the only way to extract those property names from the associative array (or JavaScript object) named `portfolio`.

8.7 Object Properties and Methods

As we discussed earlier, all objects in JavaScript inherit from the `Object` class. While more specialized classes, such as the built-in `String` class or a user-defined `Complex` class, define properties and methods of their own, all objects, whatever their class, also support the properties and methods defined by the `Object` class. Because of their universality, these properties and methods are of particular interest.

8.7.1 The constructor Property

Starting with JavaScript 1.1, every object has a `constructor` property that refers to the constructor function used to initialize the object. For example, if I create an object `o` with the `Complex()` constructor, the property `o.constructor` refers to `Complex`:

```
var o = new Complex(1,2);
o.constructor == Complex; // Evaluates to true
```

Each `Complex` object (or object of whatever type) does not have its own unique `constructor` property, of course; instead, this property is inherited from the prototype object. As discussed earlier in this chapter, JavaScript creates a prototype object for each constructor function you define and assigns that object to the `prototype` property of the constructor. What I did not reveal earlier, however, is that the prototype object is not initially empty. When created, it includes a `constructor` property that refers to the constructor function. That is, for any function `f`, `f.prototype.constructor` is always equal to `f` (unless we set it to something else).

Since the constructor function defines the class of an object, the `constructor` property can be a powerful tool for determining the type of any given object. For example, you might use code like the following to determine the type of an unknown object:

```
if ((typeof o == "object") && (o.constructor == Date))
    // Then do something with the Date object...
```

The existence of the `constructor` property is not always guaranteed, however. The author of a class might replace the prototype object of a constructor with an entirely new object, for example, and the new object might not have a valid `constructor` property.

8.7.2 The `toString()` Method

The `toString()` method takes no arguments; it returns a string that somehow represents the type and/or value of the object on which it is invoked. JavaScript invokes this method of an object whenever it needs to convert the object to a string. This occurs, for example, when you use the `+` operator to concatenate a string with an object or when you pass an object to a method such as `alert()` or `document.write()`.

The default `toString()` method is not very informative. For example, the following lines of code simply cause the browser to display the string "[object Object]":^[6]

^[6] In client-side JavaScript in Netscape, if the `language` attribute of the `<script>` tag is explicitly set to "JavaScript1.2", the `toString()` method behaves differently: it displays the names and values of all the fields of the object, using object literal notation. This violates the ECMAScript specification.

```
c = new Circle(1, 0, 0);
document.write(c);
```

Because this default method does not display much useful information, many classes define their own versions of `toString()`. For example, when an array is converted to a string, we obtain a list of the array elements, themselves each converted to a string, and when a function is converted to a string, we obtain the source code for the function.

The idea behind `toString()` is that each class of objects has its own particular string representation, so it should define an appropriate `toString()` method to convert objects to that string form. Thus, when you define a class, you should define a custom `toString()` method for it so that instances of the class can be converted to meaningful strings. The string should contain information about the object being converted, as this is useful for debugging purposes. If the string conversion is chosen carefully, it can also be useful in programs themselves.

The following code shows a `toString()` method we might define for the `Circle` class of [Example 8-5](#):

```
Circle.prototype.toString = function ( ) {
    return "[Circle of radius " + this.r + ", centered at ("
        + this.x + ", " + this.y + ").]";
}
```

With this `toString()` method defined, a typical `Circle` object might be converted to the string "[Circle of radius 1, centered at (0,0).]".

If you look back at [Example 8-6](#), you'll see that it defines a `toString()` method for our `Complex` class of complex numbers.

One interesting feature of the default `toString()` method defined by the `Object` class is that it reveals some internal type information about built-in objects. This default `toString()` method always returns a string of the form:

```
[object class]
```

`class` is the internal type of the object and usually corresponds to the name of the constructor function for the object. For example, `Array` objects have a `class` of "Array", `Function` objects have a `class` of "Function", and `Date` objects have a `class` of "Date". The built-in `Math` object has a `class` of "Math", and all `Error` objects (including instances of the various `Error` subclasses) have a `class` of "Error". Client-side JavaScript objects and any other objects defined by the JavaScript implementation have an implementation-defined `class` (such as "Window", "Document", or "Form"). User-defined objects, such as the `Circle` and `Complex` classes defined earlier in this chapter, always have a `class` of "Object".

Note that this `class` value provides useful information that is not supplied by the `typeof` operator (which returns either "Object" or "Function" for all objects). The `class` value provides information like that provided by the `constructor` property described earlier, but the `class` value provides it in the form of a string, instead of in the form of a constructor function. The only way to obtain this `class` value, however, is through the default `toString()` method defined by `Object`. Because classes often define their own versions of this method, we cannot simply invoke the `toString()` method of an object:

```
o.toString() // May invoke a customized toString( ) method for the
object
```

Instead, we have to refer explicitly to the default `toString()` function as the `Object.prototype.toString` object and use the `apply()` method of the function to invoke it on the desired object:

```
Object.prototype.toString.apply(o); // Always invokes the default
toString( )
```

We can use this technique to define a function that provides enhanced "type of" functionality:

```
// An enhanced "type of" function. Returns a string that describes the
// type of x. Note that it returns "Object" for any user-defined object
// types.
function Typeof(x) {
    // Start with the typeof operator
```

```

var t = typeof x;
// If the result is not vague, return it
if (t != "object") return t;
// Otherwise, x is an object. Get its class value to try to
// find out what kind of object it is.
var c = Object.prototype.toString.apply(x); // Returns "[object
class]"
c = c.substring(8, c.length-1);           // Strip off "[object"
and "]"
return c;
}

```

8.7.3 The `toLocaleString()` Method

In ECMAScript v3 and JavaScript 1.5, the `Object` class defines a `toLocaleString()` method in addition to its `toString()` method. The purpose of this method is to return a localized string representation of the object. The default `toLocaleString()` method defined by `Object` doesn't do any localization itself; it always return exactly the same thing as `toString()`. Subclasses, however, may define their own versions of `toLocaleString()`. In ECMAScript v3, the `Array`, `Date`, and `Number` classes do define `toLocaleString()` methods that return localized values.

8.7.4 The `valueOf()` Method

The `valueOf()` method is much like the `toString()` method, but it is called when JavaScript needs to convert an object to some primitive type other than a string -- typically, a number. Where possible, the function should return a primitive value that somehow represents the value of the object referred to by the `this` keyword.

By definition, objects are not primitive values, so most objects do not have a primitive equivalent. Thus, the default `valueOf()` method defined by the `Object` class performs no conversion and simply returns the object on which it is invoked. Classes such as `Number` and `Boolean` have obvious primitive equivalents, so they override the `valueOf()` method to return appropriate primitive values. This is why `Number` and `Boolean` objects can behave so much like their equivalent primitive values.

Occasionally, you may define a class that has some reasonable primitive equivalent. In this case, you may want to define a custom `valueOf()` method for the class. If you refer back to [Example 8-6](#), you'll see that we defined a `valueOf()` method for the `Complex` class. This method simply returned the real part of the complex number. Thus, when a `Complex` object is used in a numeric context, it behaves as if it were a real number without its imaginary component. For example, consider the following code:

```

var a = new Complex(5,4);
var b = new Complex(2,1);
var c = Complex.subtract(a,b); // c is the complex number {3,3}
var d = a - b;                // d is the number 3

```

One note of caution about defining a `valueOf()` method: the `valueOf()` method can, in some circumstances, take priority over the `toString()` method when converting an object to a string. Thus, when you define a `valueOf()` method for a class, you may need to be more explicit about calling the `toString()` method when you want to force an object of that class to be converted to a string. To continue with the Complex example:

```
alert("c = " + c);           // Uses valueOf(); displays "c = 3"
alert("c = " + c.toString()); // Displays "c = {3,3}"
```

8.7.5 The `hasOwnProperty()` Method

The `hasOwnProperty()` method returns `true` if the object locally defines a noninherited property with the name specified by the single string argument. Otherwise, it returns `false`. For example:

```
var o = new Object();
o.hasOwnProperty("undef"); // false: the property is not defined
o.hasOwnProperty("toString"); // false: toString is an inherited
property
Math.hasOwnProperty("cos"); // true: the Math object has a cos
property
```

8.7.6 The `propertyIsEnumerable()` Method

The `propertyIsEnumerable()` method returns `true` if the object defines a property with the name specified by the single string argument to the method and if that property would be enumerated by a `for/in` loop. Otherwise, it returns `false`. For example:

```
var o = { x:1 };
o.propertyIsEnumerable("x"); // true: property exists and is
enumerable
o.propertyIsEnumerable("y"); // false: property doesn't exist
o.propertyIsEnumerable("valueOf"); // false: property isn't enumerable
```

Note that the ECMAScript specification states that `propertyIsEnumerable()` considers only properties defined directly by the object, not inherited properties. This unfortunate restriction makes the function less useful, because a return value of `false` may mean either that the property is not enumerable or that it is enumerable but is an inherited property.

8.7.7 The `isPrototypeOf()` Method

The `isPrototypeOf()` method returns `true` if the object is the prototype object of the argument. Otherwise, it returns `false`. Using this method is similar to using the `constructor` property of an object. For example:

```
var o = new Object( );
Object.prototype.isPrototypeOf(o);           // true: o.constructor ==
Object                                       // false
Object.isPrototypeOf(o);                     // false
o.isPrototypeOf(Object.prototype);           // false
Function.prototype.isPrototypeOf(Object);    // true: Object.constructor
== Function
```


Chapter 9. Arrays

[Chapter 8](#) documented the JavaScript object type -- a composite data type that holds named values. This chapter documents arrays -- a composite data type that holds numbered values. Note that the arrays we'll discuss in this chapter are different from the associative arrays described in the previous chapter. Associative arrays associate values with strings. The arrays described in this chapter are just regular numeric arrays; they associate values with non-negative integers.

Throughout this book, we often treat objects and arrays as distinct data types. This is a useful and reasonable simplification; you can treat objects and arrays as separate types for most of your JavaScript programming. To fully understand the behavior of objects and arrays, however, you have to know the truth: an array is nothing more than an object with a thin layer of extra functionality. We see this when we use the `typeof` operator: applied to an array value, it returns the string "object". Note that the extra functionality of arrays was introduced in JavaScript 1.1. Arrays are not supported in JavaScript 1.0.

This chapter documents basic array syntax, array programming techniques, and methods that operate on arrays.

9.1 Arrays and Array Elements

An *array* is a data type that contains or stores numbered values. Each numbered value is called an *element* of the array, and the number assigned to an element is called its *index*. Because JavaScript is an untyped language, an element of an array may be of any type, and different elements of the same array may be of different types. Array elements may even contain other arrays, which allows you to create data structures that are arrays of arrays.

9.1.1 Creating Arrays

In JavaScript 1.1 and later, arrays are created with the `Array()` constructor and the `new` operator. You can invoke the `Array()` constructor in three distinct ways.

The first way is to call it with no arguments:

```
var a = new Array( );
```

This method creates an empty array with no elements.

The second method of invoking the `Array()` constructor allows you to explicitly specify values for the first n elements of an array:

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

In this form, the constructor takes a list of arguments. Each argument specifies an element value and may be of any type. Elements are assigned to the array starting with element 0. The `length` property of the array is set to the number of arguments passed to the constructor.

The third way to invoke the `Array()` constructor is to call it with a single numeric argument, which specifies a length:

```
var a = new Array(10);
```

This technique creates an array with the specified number of elements (each of which has the `undefined` value) and sets the array's `length` property to the value specified.^[1]

^[1] In client-side JavaScript in Netscape, if the `language` attribute of the `<script>` tag is explicitly set to "JavaScript1.2", this third form of the `Array()` constructor behaves like the second form: it creates an array of length one and initializes that array element to the constructor argument. This does not conform to the ECMAScript standard.

Finally, array literals provide another way to create arrays. An array literal allows us to embed an array value directly into a JavaScript program in the same way that we define a string literal by placing the string text between quotation marks. To create an array literal, simply place a comma-separated list of values between square brackets. For example:

```
var primes = [2, 3, 5, 7, 11];  
var a = ['a', true, 4.78];
```

Array literals can contain object literals or other array literals:

```
var b = [[1, {x:1, y:2}], [2, {x:3, y:4}]];
```

[Chapter 3](#) provides complete details on array literals.

9.1.2 Reading and Writing Array Elements

You access an element of an array using the `[]` operator. A reference to the array should appear to the left of the brackets. An arbitrary expression that has a non-negative integer value should be inside the brackets. You can use this syntax to both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
value = a[0];  
a[1] = 3.14;  
i = 2;  
a[i] = 3;  
a[i + 1] = "hello";  
a[a[i]] = a[0];
```

In some languages, the first element of an array is at index 1. In JavaScript (as in C, C++, and Java), however, the first element of an array is at index 0.

As we saw in [Chapter 8](#), the `[]` operator can also be used to access named object properties:

```
my['salary'] *= 2;
```

This is a clue that tells us that objects and arrays are fundamentally the same thing.

Note that array indexes must be integers greater than or equal to 0 and less than $2^{32} - 1$. If you use a number that is too large, a negative number, or a floating-point number (or a boolean, an object, or other value), JavaScript converts it to a string and uses the resulting string as the name of an object property, not as an array index. Thus, the following line creates a new property named "-1.23"; it does not define a new array element:

```
a[-1.23] = true;
```

9.1.3 Adding New Elements to an Array

In languages such as C and Java, an array has a fixed number of elements that must be specified when you create the array. This is not the case in JavaScript -- an array can have any number of elements, and you can change the number of elements at any time.

To add a new element to an array, simply assign a value to it:

```
a[10] = 10;
```

Arrays in JavaScript may be *sparse*. This means that array indexes need not fall into a contiguous range of numbers; a JavaScript implementation may allocate memory only for those array elements that are actually stored in the array. Thus, when you execute the following lines of code, the JavaScript interpreter will typically allocate memory only for array indexes 0 and 10,000, not for the 9,999 indexes between:

```
a[0] = 1;  
a[10000] = "this is element 10,000";
```

Note that array elements can also be added to objects:

```
var c = new Circle(1,2,3);  
c[0] = "this is an array element of an object!"
```

This example merely defines a new object property named "0", however. Adding array elements to an object does not make it an array. Arrays created with the `Array()` constructor or an array literal have some special features, explained below, that objects do not share.

9.1.4 Array Length

All arrays, whether created with the `Array()` constructor or defined with an array literal, have a special `length` property that specifies how many elements the array contains. More precisely, since arrays can have undefined elements, the `length` property is *always* one larger than the largest element number in the array. Unlike regular object properties, the `length` property of an array is automatically updated to maintain this invariant when new elements are added to the array. The following code illustrates:

```
var a = new Array( ); // a.length == 0 (no elements defined)
a = new Array(10); // a.length == 10 (empty elements 0-9 defined)
a = new Array(1,2,3); // a.length == 3 (elements 0-2 defined)
a = [4, 5]; // a.length == 2 (elements 0 and 1 defined)
a[5] = -1; // a.length == 6 (elements 0, 1, and 5 defined)
a[49] = 0; // a.length == 50 (elements 0, 1, 5, and 49
defined)
```

Remember that array indexes must be less than $2^{32} - 1$, which means that the largest possible value for the `length` property is $2^{32} - 1$.

Probably the most common use of the `length` property of an array is to allow us to loop through the elements of an array:

```
var fruits = ["mango", "banana", "cherry", "pear"];
for(var i = 0; i < fruits.length; i++)
    alert(fruits[i]);
```

This example assumes, of course, that elements of the array are contiguous and begin at element 0. If this were not the case, we would want to test that each array element was defined before using it:

```
for(var i = 0; i < fruits.length; i++)
    if (fruits[i] != undefined) alert(fruits[i]);
```

The `length` property of an array is a read/write value. If you set `length` to a value smaller than its current value, the array is truncated to the new length; any elements that no longer fit are discarded and their values are lost. If you make `length` larger than its current value, new, undefined elements are added at the end of the array to increase it to the newly specified size.

Truncating an array by setting its `length` property is the only way that you can actually shorten an array. If you use the `delete` operator to delete an array element, that element becomes undefined, but the `length` property does not change.

Note that although objects can be assigned array elements, they do not have a `length` property. The `length` property, with its special behavior, is the most important feature of arrays. The other features that make arrays different from objects are the various methods defined by the Array class, which are described in [Section 9.2](#).

9.1.5 Multidimensional Arrays

JavaScript does not support true multidimensional arrays, but it does allow you to approximate them quite nicely with arrays of arrays. To access a data element in an array of arrays, simply use the `[]` operator twice. For example, suppose the variable `matrix` is an array of arrays of numbers. Every element `matrix[x]` is an array of numbers. To access a particular number within this array, you would write `matrix[x][y]`.

9.2 Array Methods

In addition to the `[]` operator, arrays can be manipulated through various methods provided by the Array class. The following sections introduce these methods. Many of the methods were inspired in part by the Perl programming language; Perl programmers may find them comfortably familiar. As usual, this is an overview only; complete details can be found in the core reference section of this book.

9.2.1 `join()`

The `Array.join()` method converts all the elements of an array to strings and concatenates them. You can specify an optional string that is used to separate the elements in the resulting string. If no separator string is specified, a comma is used. For example, the following lines of code produce the string "1,2,3":

```
var a = [1, 2, 3];    // Create a new array with these three elements
var s = a.join( );   // s == "1,2,3"
```

The following invocation specifies the optional separator to produce a slightly different result:

```
s = a.join(", ");    // s == "1, 2, 3"
```

Notice the space after the comma. The `Array.join()` method is the inverse of the `String.split()` method, which creates an array by breaking up a string into pieces.

9.2.2 reverse()

The `Array.reverse()` method reverses the order of the elements of an array and returns the reversed array. It does this in place -- in other words, it doesn't create a new array with the elements rearranged, but instead rearranges them in the already existing array. For example, the following code, which uses the `reverse()` and `join()` methods, produces the string "3,2,1":

```
var a = new Array(1,2,3);    // a[0] = 1, a[1] = 2, a[2] = 3
a.reverse( );              // now a[0] = 3, a[1] = 2, a[2] = 1
var s = a.join( );         // s == "3,2,1"
```

9.2.3 sort()

`Array.sort()` sorts the elements of an array in place and returns the sorted array. When `sort()` is called with no arguments, it sorts the array elements in alphabetical order (temporarily converting them to strings to perform the comparison, if necessary):

```
var a = new Array("banana", "cherry", "apple");
a.sort( );
var s = a.join(", "); // s == "apple, banana, cherry"
```

If an array contains undefined elements, they are sorted to the end of the array.

To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to `sort()`. This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0. So, for example, to sort array elements into numerical rather than alphabetical order, you might do this:

```
var a = [33, 4, 1111, 222];
a.sort( ); // Alphabetical order: 1111, 222, 33, 4
a.sort(function(a,b) { // Numerical order: 4, 33, 222, 1111
    return a-b; // Returns < 0, 0, or > 0, depending on order
});
```

Note the convenient use of a function literal in this code. Since the comparison function is used only once, there is no need to give it a name.

As another example of sorting array items, you might perform a case-insensitive alphabetical sort on an array of strings by passing a comparison function that converts both of its arguments to lowercase (with the `toLowerCase()` method) before comparing

them. You can probably think of other comparison functions that sort numbers into various esoteric orders: reverse numerical order, odd numbers before even numbers, etc. The possibilities become more interesting, of course, when the array elements you are comparing are objects, rather than simple types like numbers or strings.

9.2.4 `concat()`

The `Array.concat()` method creates and returns a new array that contains the elements of the original array on which `concat()` was invoked, followed by each of the arguments to `concat()`. If any of these arguments is itself an array, it is flattened and its elements are added to the returned array. Note, however, that `concat()` does not recursively flatten arrays of arrays. Here are some examples:

```
var a = [1,2,3];
a.concat(4, 5)           // Returns [1,2,3,4,5]
a.concat([4,5]);        // Returns [1,2,3,4,5]
a.concat([4,5],[6,7])   // Returns [1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]])  // Returns [1,2,3,4,5,[6,7]]
```

9.2.5 `slice()`

The `Array.slice()` method returns a *slice*, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned. The returned array contains the element specified by the first argument and all subsequent elements up to, but not including, the element specified by the second argument. If only one argument is specified, the returned array contains all elements from the start position to the end of the array. If either argument is negative, it specifies an array element relative to the last element in the array. An argument of `-1`, for example, specifies the last element in the array, and an argument of `-3` specifies the third from last element of the array. Here are some examples:

```
var a = [1,2,3,4,5];
a.slice(0,3);           // Returns [1,2,3]
a.slice(3);             // Returns [4,5]
a.slice(1,-1);         // Returns [2,3,4]
a.slice(-3,-2);        // Returns [3]
```

9.2.6 `splice()`

The `Array.splice()` method is a general-purpose method for inserting or removing elements from an array. `splice()` modifies the array in place; it does not return a new array, as `slice()` and `concat()` do. Note that `splice()` and `slice()` have very similar names but perform substantially different operations.

`splice()` can delete elements from an array, insert new elements into an array, or perform both operations at the same time. Array elements that appear after the insertion

or deletion are moved as necessary so that they remain contiguous with the rest of the array. The first argument to `splice()` specifies the array position at which the insertion and/or deletion is to begin. The second argument specifies the number of elements that should be deleted from (spliced out of) the array. If this second argument is omitted, all array elements from the start element to the end of the array are removed. `splice()` returns an array of the deleted elements, or an empty array if no elements were deleted. For example:

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);    // Returns [5,6,7,8]; a is [1,2,3,4]
a.splice(1,2); // Returns [2,3]; a is [1,4]
a.splice(1,1); // Returns [4]; a is [1]
```

The first two arguments to `splice()` specify which array elements are to be deleted. These arguments may be followed by any number of additional arguments that specify elements to be inserted into the array, starting at the position specified by the first argument. For example:

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b'); // Returns []; a is [1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3); // Returns ['a','b']; a is [1,2,[1,2],3,3,4,5]
```

Note that, unlike `concat()`, `splice()` does not flatten array arguments that it inserts. That is, if it is passed an array to insert, it inserts the array itself, not the elements of that array.

9.2.7 `push()` and `pop()`

The `push()` and `pop()` methods allow us to work with arrays as if they were stacks. The `push()` method appends one or more new elements to the end of an array and returns the new length of the array.^[2] The `pop()` method does the reverse: it deletes the last element of an array, decrements the array length, and returns the value that it removed. Note that both of these methods modify the array in place rather than producing a modified copy of the array. The combination of `push()` and `pop()` allows us to use a JavaScript array to implement a first in, last out stack. For example:

^[2] In Netscape, when the `language` attribute of the `<script>` tag is explicitly set to "JavaScript1.2", `push()` returns the last value it appends to the array, rather than the new length of the array.

```
var stack = [];           // stack: []
stack.push(1,2);         // stack: [1,2]   Returns 2
stack.pop( );           // stack: [1]     Returns 2
stack.push(3);          // stack: [1,3]   Returns 2
stack.pop( );           // stack: [1]     Returns 3
stack.push([4,5]);      // stack: [1,[4,5]] Returns 2
stack.pop( );           // stack: [1]     Returns [4,5]
stack.pop( );           // stack: []      Returns 1
```


9.2.8 unshift() and shift()

The `unshift()` and `shift()` methods behave much like `push()` and `pop()`, except that they insert and remove elements from the beginning of an array, rather than from the end. `unshift()` adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the new length of the array. `shift()` removes and returns the first element of the array, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array. For example:

```
var a = [];           // a:[]
a.unshift(1);        // a:[1]       Returns: 1
a.unshift(22);       // a:[22,1]    Returns: 2
a.shift( );          // a:[1]       Returns: 22
a.unshift(3,[4,5]);  // a:[3,[4,5],1] Returns: 3
a.shift( );          // a:[[4,5],1] Returns: 3
a.shift( );          // a:[1]       Returns: [4,5]
a.shift( );          // a:[]       Returns: 1
```

Note the possibly surprising behavior of `unshift()` when it's invoked with multiple arguments. Instead of being inserted into the array one at a time, arguments are inserted all at once (as with the `splice()` method). This means that they appear in the resulting array in the same order in which they appeared in the argument list. Had the elements been inserted one at a time, their order would have been reversed.

9.2.9 toString() and toLocaleString()

An array, like any JavaScript object, has a `toString()` method. For an array, this method converts each of its elements to a string (calling the `toString()` methods of its elements, if necessary) and outputs a comma-separated list of those strings. Note that the output does not include square brackets or any other sort of delimiter around the array value. For example:

```
[1,2,3].toString( ) // Yields '1,2,3'
["a", "b", "c"].toString( ) // Yields 'a,b,c'
[1, [2, 'c']].toString( ) // Yields '1,2,c'
```

Note that `toString()` returns the same string the `join()` method does when it is invoked with no arguments.^[3]

^[3] In Netscape, when the `language` attribute of the `<script>` tag is set to "JavaScript1.2", `toString()` behaves in a more complex way. In this case, it converts arrays to strings that include square brackets, and includes quotation marks around array elements that are strings, so that the resulting strings are valid array literal expressions.

`toLocaleString()` is the localized version of `toString()`. It converts each array element to a string by calling the `toLocaleString()` method of the element, and then it concatenates the resulting strings using a locale-specific (and implementation-defined) separator string.

Chapter 10. Pattern Matching with Regular Expressions

A *regular expression* is an object that describes a pattern of characters. The JavaScript `RegExp` class represents regular expressions, and both `String` and `RegExp` define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.^[1]

^[1] The term "regular expression" is an obscure one that dates back many years. The syntax used to describe a textual pattern is indeed a type of expression. However, as we'll see, that syntax is far from regular! A regular expression is sometimes called a "regexp" or even an "RE."

JavaScript regular expressions were standardized in ECMAScript v3. JavaScript 1.2 implements a subset of the regular expression features required by ECMAScript v3, and JavaScript 1.5 implements the full standard. JavaScript regular expressions are strongly based on the regular expression facilities of the Perl programming language. Roughly speaking, we can say that JavaScript 1.2 implements Perl 4 regular expressions, and JavaScript 1.5 implements a large subset of Perl 5 regular expressions.

This chapter begins by defining the syntax that regular expressions use to describe textual patterns. Then it moves on to describe the `String` and `RegExp` methods that use regular expressions.

10.1 Defining Regular Expressions

In JavaScript, regular expressions are represented by `RegExp` objects. `RegExp` objects may be created with the `RegExp()` constructor, of course, but they are more often created using a special literal syntax. Just as string literals are specified as characters within quotation marks, regular expression literals are specified as characters within a pair of slash (`/`) characters. Thus, your JavaScript code may contain lines like this:

```
var pattern = /s$/;
```

This line creates a new `RegExp` object and assigns it to the variable `pattern`. This particular `RegExp` object matches any string that ends with the letter "s". (We'll talk about the grammar for defining patterns shortly.) This regular expression could have equivalently been defined with the `RegExp()` constructor like this:

```
var pattern = new RegExp("s$");
```

Creating a `RegExp` object, either literally or with the `RegExp()` constructor, is the easy part. The more difficult task is describing the desired pattern of characters using regular

expression syntax. JavaScript adopts a fairly complete subset of the regular expression syntax used by Perl, so if you are an experienced Perl programmer, you already know how to describe patterns in JavaScript.

Regular expression pattern specifications consist of a series of characters. Most characters, including all alphanumeric characters, simply describe characters to be matched literally. Thus, the regular expression `/java/` matches any string that contains the substring "java". Other characters in regular expressions are not matched literally, but have special significance. For example, the regular expression `/s$/` contains two characters. The first, "s", matches itself literally. The second, "\$", is a special metacharacter that matches the end of a string. Thus, this regular expression matches any string that contains the letter "s" as its last character.

The following sections describe the various characters and metacharacters used in JavaScript regular expressions. Note, however, that a complete tutorial on regular expression grammar is beyond the scope of this book. For complete details of the syntax, consult a book on Perl, such as *Programming Perl*, by Larry Wall, Tom Christiansen, and Jon Orwant (O'Reilly). *Mastering Regular Expressions*, by Jeffrey E.F. Friedl (O'Reilly), is another excellent source of information on regular expressions.

10.1.1 Literal Characters

As we've seen, all alphabetic characters and digits match themselves literally in regular expressions. JavaScript regular expression syntax also supports certain nonalphabetic characters through escape sequences that begin with a backslash (`\`). For example, the sequence `\n` matches a literal newline character in a string. [Table 10-1](#) lists these characters.

Table 10-1. Regular expression literal characters	
Character	Matches
Alphanumeric character	Itself
<code>\0</code>	The NUL character (<code>\u0000</code>)
<code>\t</code>	Tab (<code>\u0009</code>)
<code>\n</code>	Newline (<code>\u000A</code>)
<code>\v</code>	Vertical tab (<code>\u000B</code>)
<code>\f</code>	Form feed (<code>\u000C</code>)
<code>\r</code>	Carriage return (<code>\u000D</code>)
<code>\xnn</code>	The Latin character specified by the hexadecimal number <i>nn</i> ; for

Table 10-1. Regular expression literal characters

Character	Matches
	example, <code>\x0A</code> is the same as <code>\n</code>
<code>\uxxxx</code>	The Unicode character specified by the hexadecimal number <code>xxxxx</code> ; for example, <code>\u0009</code> is the same as <code>\t</code>
<code>\cX</code>	The control character <code>^X</code> ; for example, <code>\cJ</code> is equivalent to the newline character <code>\n</code>

A number of punctuation characters have special meanings in regular expressions. They are:

`^ $. * + ? = ! : | \ / () [] { }`

We'll learn the meanings of these characters in the sections that follow. Some of these characters have special meaning only within certain contexts of a regular expression and are treated literally in other contexts. As a general rule, however, if you want to include any of these punctuation characters literally in a regular expression, you must precede them with a `\`. Other punctuation characters, such as quotation marks and `@`, do not have special meaning and simply match themselves literally in a regular expression.

If you can't remember exactly which punctuation characters need to be escaped with a backslash, you may safely place a backslash before any punctuation character. On the other hand, note that many letters and numbers have special meaning when preceded by a backslash, so any letters or numbers that you want to match literally should not be escaped with a backslash. To include a backslash character literally in a regular expression, you must escape it with a backslash, of course. For example, the following regular expression matches any string that includes a backslash: `/\\`.

10.1.2 Character Classes

Individual literal characters can be combined into *character classes* by placing them within square brackets. A character class matches any one character that is contained within it. Thus, the regular expression `/[abc]/` matches any one of the letters a, b, or c. Negated character classes can also be defined -- these match any character except those contained within the brackets. A negated character class is specified by placing a caret (^) as the first character inside the left bracket. The regexp `/[^abc]/` matches any one character other than a, b, or c. Character classes can use a hyphen to indicate a range of characters. To match any one lowercase character from the Latin alphabet, use `/[a-z]/`, and to match any letter or digit from the Latin alphabet, use `/[a-zA-Z0-9]/`.

Because certain character classes are commonly used, the JavaScript regular expression syntax includes special characters and escape sequences to represent these common classes. For example, `\s` matches the space character, the tab character, and any other Unicode whitespace character, and `\S` matches any character that is *not* Unicode whitespace. [Table 10-2](#) lists these characters and summarizes character class syntax. (Note that several of these character class escape sequences match only ASCII characters and have not been extended to work with Unicode characters. You can explicitly define your own Unicode character classes; for example, `/[\u0400-04FF]/` matches any one Cyrillic character.)

Table 10-2. Regular expression character classes	
Character	Matches
<code>[...]</code>	Any one character between the brackets.
<code>[^...]</code>	Any one character not between the brackets.
<code>.</code>	Any character except newline or another Unicode line terminator.
<code>\w</code>	Any ASCII word character. Equivalent to <code>[a-zA-Z0-9_]</code> .
<code>\W</code>	Any character that is not an ASCII word character. Equivalent to <code>[^a-zA-Z0-9_]</code> .
<code>\s</code>	Any Unicode whitespace character.
<code>\S</code>	Any character that is not Unicode whitespace. Note that <code>\w</code> and <code>\S</code> are not the same thing.
<code>\d</code>	Any ASCII digit. Equivalent to <code>[0-9]</code> .
<code>\D</code>	Any character other than an ASCII digit. Equivalent to <code>[^0-9]</code> .
<code>[\b]</code>	A literal backspace (special case).

Note that the special character class escapes can be used within square brackets. `\s` matches any whitespace character and `\d` matches any digit, so `/[\s\d]/` matches any one whitespace character or digit. Note that there is one special case. As we'll see later, the `\b` escape has a special meaning. When used within a character class, however, it represents the backspace character. Thus, to represent a backspace character literally in a regular expression, use the character class with one element: `/[\b]/`.

10.1.3 Repetition

With the regular expression syntax we have learned so far, we can describe a two-digit number as `/\d\d/` and a four-digit number as `/\d\d\d\d/`. But we don't have any way to describe, for example, a number that can have any number of digits or a string of three

letters followed by an optional digit. These more complex patterns use regular expression syntax that specifies how many times an element of a regular expression may be repeated.

The characters that specify repetition always follow the pattern to which they are being applied. Because certain types of repetition are quite commonly used, there are special characters to represent these cases. For example, `+` matches one or more occurrences of the previous pattern. [Table 10-3](#) summarizes the repetition syntax. The following lines show some examples:

```
/\d{2,4}/      // Match between two and four digits
/\w{3}\d?/    // Match exactly three word characters and an optional
digit
/\s+java\s+/  // Match "java" with one or more spaces before and after
/[^"]*/       // Match zero or more non-quote characters
```

Table 10-3. Regular expression repetition characters

Character	Meaning
<code>{n,m}</code>	Match the previous item at least <i>n</i> times but no more than <i>m</i> times.
<code>{n,}</code>	Match the previous item <i>n</i> or more times.
<code>{n}</code>	Match exactly <i>n</i> occurrences of the previous item.
<code>?</code>	Match zero or one occurrences of the previous item. That is, the previous item is optional. Equivalent to <code>{0,1}</code> .
<code>+</code>	Match one or more occurrences of the previous item. Equivalent to <code>{1,}</code> .
<code>*</code>	Match zero or more occurrences of the previous item. Equivalent to <code>{0,}</code> .

Be careful when using the `*` and `?` repetition characters. Since these characters may match zero instances of whatever precedes them, they are allowed to match nothing. For example, the regular expression `/a*/` actually matches the string "bbbb", because the string contains zero occurrences of the letter a!

10.1.3.1 Non-greedy repetition

The repetition characters listed in [Table 10-3](#) match as many times as possible while still allowing any following parts of the regular expression to match. We say that the repetition is "greedy." It is also possible (in JavaScript 1.5 and later -- this is one of the Perl 5 features not implemented in JavaScript 1.2) to specify that repetition should be done in a non-greedy way. Simply follow the repetition character or characters with a question mark: `??`, `+`, `*?`, or even `{1,5}?`. For example, the regular expression `/a+/?` matches one or more occurrences of the letter a. When applied to the string "aaa", it matches all three letters. But `/a+?/?` matches one or more occurrences of the letter a,

matching as few characters as necessary. When applied to the same string, this pattern matches only the first letter a.

Using non-greedy repetition may not always produce the results you expect. Consider the pattern `/a*b/`, which matches zero or more letters a followed by the letter b. When applied to the string "aaab", it matches the entire string. Now let's use the non-greedy version: `/a*b?/`. This should match the letter b preceded by the fewest number of a's possible. When applied to the same string "aaab", you might expect it to match only the last letter b. In fact, however, this pattern matches the entire string as well, just like the greedy version of the pattern. This is because regular expression pattern matching is done by finding the first position in the string at which a match is possible. The non-greedy version of our pattern does match at the first character of the string, so this match is returned; matches at subsequent characters are never even considered.

10.1.4 Alternation, Grouping, and References

The regular expression grammar includes special characters for specifying alternatives, grouping subexpressions, and referring to previous subexpressions. The `|` character separates alternatives. For example, `/ab|cd|ef/` matches the string "ab" or the string "cd" or the string "ef". And `/\d{3}|[a-z]{4}/` matches either three digits or four lowercase letters.

Note that alternatives are considered left to right until a match is found. If the left alternative matches, the right alternative is ignored, even if it would have produced a "better" match. Thus, when the pattern `/a|ab/` is applied to the string "ab", it matches only the first letter.

Parentheses have several purposes in regular expressions. One purpose is to group separate items into a single subexpression, so that the items can be treated as a single unit by `|`, `*`, `+`, `?`, and so on. For example, `/java(script)?/` matches "java" followed by the optional "script". And `/(ab|cd)+|ef/` matches either the string "ef" or one or more repetitions of either of the strings "ab" or "cd".

Another purpose of parentheses in regular expressions is to define subpatterns within the complete pattern. When a regular expression is successfully matched against a target string, it is possible to extract the portions of the target string that matched any particular parenthesized subpattern. (We'll see how these matching substrings are obtained later in the chapter.) For example, suppose we are looking for one or more lowercase letters followed by one or more digits. We might use the pattern `/[a-z]+\d+/`. But suppose we only really care about the digits at the end of each match. If we put that part of the pattern in parentheses (`/[a-z]+(\d+)/`), we can extract the digits from any matches we find, as explained later.

A related use of parenthesized subexpressions is to allow us to refer back to a subexpression later in the same regular expression. This is done by following a `\` character by a digit or digits. The digits refer to the position of the parenthesized

subexpression within the regular expression. For example, `\1` refers back to the first subexpression and `\3` refers to the third. Note that, because subexpressions can be nested within others, it is the position of the left parenthesis that is counted. In the following regular expression, for example, the nested subexpression `([Ss]cript)` is referred to as `\2`:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

A reference to a previous subexpression of a regular expression does *not* refer to the pattern for that subexpression, but rather to the text that matched the pattern. Thus, references can be used to enforce a constraint that separate portions of a string contain exactly the same characters. For example, the following regular expression matches zero or more characters within single or double quotes. However, it does not require the opening and closing quotes to match (i.e., both single quotes or both double quotes):

```
/[ '" ] [ ^ ' " ] * [ ' " ] /
```

To require the quotes to match, we can use a reference:

```
/([ '" ] ) [ ^ \1 ] * \1 /
```

The `\1` matches whatever the first parenthesized subexpression matched. In this example, it enforces the constraint that the closing quote match the opening quote. This regular expression does not allow single quotes within double-quoted strings or vice versa. It is not legal to use a reference within a character class, so we cannot write:

```
/([ '" ] ) [ ^ \1 ] * \1 /
```

Later in this chapter, we'll see that this kind of reference to a parenthesized sub-expression is a powerful feature of regular expression search-and-replace operations.

In JavaScript 1.5 (but not JavaScript 1.2), it is possible to group items in a regular expression without creating a numbered reference to those items. Instead of simply grouping the items within `(` and `)`, begin the group with `(?:` and end it with `)`. Consider the following pattern, for example:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

Here, the subexpression `(?:[Ss]cript)` is used simply for grouping, so the `?` repetition character can be applied to the group. These modified parentheses do not produce a reference, so in this regular expression, `\2` refers to the text matched by `(fun\w*)`.

[Table 10-4](#) summarizes the regular expression alternation, grouping, and referencing operators.

Table 10-4. Regular expression alternation, grouping, and reference characters	
Character	Meaning
	Alternation. Match either the subexpressions to the left or the subexpression to the right.
(...)	Grouping. Group items into a single unit that can be used with *, +, ?, , and so on. Also remember the characters that match this group for use with later references.
(?:...)	Grouping only. Group items into a single unit, but do not remember the characters that match this group.
\n	Match the same characters that were matched when group number <i>n</i> was first matched. Groups are subexpressions within (possibly nested) parentheses. Group numbers are assigned by counting left parentheses from left to right. Groups formed with (?: are not numbered.

10.1.5 Specifying Match Position

We've seen that many elements of a regular expression match a single character in a string. For example, `\s` matches a single character of whitespace. Other regular expression elements match the positions between characters, instead of actual characters. `\b`, for example, matches a word boundary -- the boundary between a `\w` (ASCII word character) and a `\W` (non-word character), or the boundary between an ASCII word character and the beginning or end of a string.^[2] Elements like `\b` do not specify any characters to be used in a matched string; what they do specify, however, is legal positions at which a match can occur. Sometimes these elements are called regular expression anchors, because they anchor the pattern to a specific position in the search string. The most commonly used anchor elements are `^`, which ties the pattern to the beginning of the string, and `$`, which anchors the pattern to the end of the string.

^[2] Except within a character class (square brackets), where `\b` matches the backspace character.

For example, to match the word "JavaScript" on a line by itself, we could use the regular expression `/^JavaScript$/`. If we wanted to search for "Java" used as a word by itself (not as a prefix, as it is in "JavaScript"), we might try the pattern `/\sJava\s/`, which requires a space before and after the word. But there are two problems with this solution. First, it does not match "Java" if that word appears at the beginning or the end of a string, but only if it appears with space on either side. Second, when this pattern does find a match, the matched string it returns has leading and trailing spaces, which is not quite

what we want. So instead of matching actual space characters with `\s`, we instead match (or anchor to) word boundaries with `\b`. The resulting expression is `/\bJava\b/`. The element `\B` anchors the match to a location that is not a word boundary. Thus, the pattern `/\B[Sc]ript/` matches "JavaScript" and "postscript", but not "script" or "Scripting".

In JavaScript 1.5 (but not JavaScript 1.2), you can also use arbitrary regular expressions as anchor conditions. If you include an expression within `(?=` and `)` characters, it is a look-ahead assertion, and it specifies that the following characters must match, without actually matching them. For example, to match the name of a common programming language, but only if it is followed by a colon, you could use

`/[Jj]ava([Sc]ript)?(?=\:)/`. This pattern matches the word "JavaScript" in "JavaScript: The Definitive Guide", but it does not match "Java" in "Java in a Nutshell" because it is not followed by a colon.

If you instead introduce an assertion with `(?!`, it is a negative look-ahead assertion, which specifies that the following characters must not match. For example,

`/Java(?!Script)([A-Z]\w*)/` matches "Java" followed by a capital letter and any number of additional ASCII word characters, as long as "Java" is not followed by "Script". It matches "JavaBeans" but not "Javanese", and it matches "JavaScrip" but not "JavaScript" or "JavaScripter".

[Table 10-5](#) summarizes regular expression anchors.

Table 10-5. Regular expression anchor characters	
Character	Meaning
<code>^</code>	Match the beginning of the string and, in multiline searches, the beginning of a line.
<code>\$</code>	Match the end of the string and, in multiline searches, the end of a line.
<code>\b</code>	Match a word boundary. That is, match the position between a <code>\w</code> character and a <code>\W</code> character or between a <code>\w</code> character and the beginning or end of a string. (Note, however, that <code>[\b]</code> matches backspace.)
<code>\B</code>	Match a position that is not a word boundary.
<code>(?=p)</code>	A positive look-ahead assertion. Require that the following characters match the pattern <code>p</code> , but do not include those characters in the match.
<code>(?!p)</code>	A negative look-ahead assertion. Require that the following characters do not match the pattern <code>p</code> .

10.1.6 Flags

There is one final element of regular expression grammar. Regular expression flags specify high-level pattern-matching rules. Unlike the rest of regular expression syntax, flags are specified outside of the `/` characters; instead of appearing within the slashes, they appear following the second slash. JavaScript 1.2 supports two flags. The `i` flag specifies that pattern matching should be case-insensitive. The `g` flag specifies that pattern matching should be global -- that is, all matches within the searched string should be found. Both flags may be combined to perform a global case-insensitive match.

For example, to do a case-insensitive search for the first occurrence of the word "java" (or "Java", "JAVA", etc.), we could use the case-insensitive regular expression `/\bjava\b/i`. And to find all occurrences of the word in a string, we would add the `g` flag: `/\bjava\b/gi`.

JavaScript 1.5 supports an additional flag: `m`. The `m` flag performs pattern matching in multiline mode. In this mode, if the string to be searched contains newlines, the `^` and `$` anchors match the beginning and end of a line in addition to matching the beginning and end of a string. For example, the pattern `/Java$/im` matches "java" as well as "Java\nis fun".

[Table 10-6](#) summarizes these regular expression flags. Note that we'll see more about the `g` flag later in this chapter, when we consider the `String` and `RegExp` methods used to actually perform matches.

<i>Table 10-6. Regular expression flags</i>	
Character	Meaning
<code>i</code>	Perform case-insensitive matching.
<code>g</code>	Perform a global match. That is, find all matches rather than stopping after the first match.
<code>m</code>	Multiline mode. <code>^</code> matches beginning of line or beginning of string, and <code>\$</code> matches end of line or end of string.

10.1.7 Perl RegExp Features Not Supported in JavaScript

We've said that ECMAScript v3 specifies a relatively complete subset of the regular expression facilities from Perl 5. Advanced Perl features that are not supported by ECMAScript include the following:

- The `s` (single-line mode) and `x` (extended syntax) flags
- The `\a`, `\e`, `\l`, `\u`, `\L`, `\U`, `\E`, `\Q`, `\A`, `\Z`, `\z`, and `\G` escape sequences

- The `(?<=` positive look-behind anchor and the `(?<!` negative look-behind anchor
- The `(?#` comment and the other extended `(?` syntaxes

10.2 String Methods for Pattern Matching

Until now, we've been discussing the grammar used to create regular expressions, but we haven't examined how those regular expressions can actually be used in JavaScript code. In this section, we discuss methods of the `String` object that use regular expressions to perform pattern matching and search-and-replace operations. In the sections that follow this one, we'll continue the discussion of pattern matching with JavaScript regular expressions by discussing the `RegExp` object and its methods and properties. Note that the discussion that follows is merely an overview of the various methods and properties related to regular expressions. As usual, complete details can be found in the core reference section of this book.

Strings support four methods that make use of regular expressions. The simplest is `search()`. This method takes a regular expression argument and returns either the character position of the start of the first matching substring, or `-1` if there is no match. For example, the following call returns `4`:

```
"JavaScript".search(/script/i);
```

If the argument to `search()` is not a regular expression, it is first converted to one by passing it to the `RegExp` constructor. `search()` does not support global searches -- it ignores the `g` flag of its regular expression argument.

The `replace()` method performs a search-and-replace operation. It takes a regular expression as its first argument and a replacement string as its second argument. It searches the string on which it is called for matches with the specified pattern. If the regular expression has the `g` flag set, the `replace()` method replaces all matches in the string with the replacement string; otherwise, it replaces only the first match it finds. If the first argument to `replace()` is a string rather than a regular expression, the method searches for that string literally rather than converting it to a regular expression with the `RegExp()` constructor, as `search()` does. As an example, we could use `replace()` as follows to provide uniform capitalization of the word "JavaScript" throughout a string of text:

```
// No matter how it is capitalized, replace it with the correct
capitalization
text.replace(/javascript/gi, "JavaScript");
```

`replace()` is more powerful than this, however. Recall that parenthesized subexpressions of a regular expression are numbered from left to right and that the regular expression remembers the text that each subexpression matches. If a `$` followed

by a digit appears in the replacement string, `replace()` replaces those two characters with the text that matched the specified subexpression. This is a very useful feature. We can use it, for example, to replace straight quotes in a string with curly quotes, simulated with ASCII characters:

```
// A quote is a quotation mark, followed by any number of
// non-quotation-mark characters (which we remember), followed
// by another quotation mark.
var quote = /"([\^"]*)"/g;
// Replace the straight quotation marks with "curly quotes,"
// and leave the contents of the quote (stored in $1) unchanged.
text.replace(quote, "``$1'`');
```

The `replace()` method has other important features as well, which are described in the "String.replace()" reference page in the core reference section. Most notably, the second argument to `replace()` can be a function that dynamically computes the replacement string.

The `match()` method is the most general of the String regular expression methods. It takes a regular expression as its only argument (or converts its argument to a regular expression by passing it to the `RegExp()` constructor) and returns an array that contains the results of the match. If the regular expression has the `g` flag set, the method returns an array of all matches that appear in the string. For example:

```
"1 plus 2 equals 3".match(/\d+/g) // returns ["1", "2", "3"]
```

If the regular expression does not have the `g` flag set, `match()` does not do a global search; it simply searches for the first match. However, `match()` returns an array even when it does not perform a global search. In this case, the first element of the array is the matching string, and any remaining elements are the parenthesized subexpressions of the regular expression. Thus, if `match()` returns an array `a`, `a[0]` contains the complete match, `a[1]` contains the substring that matched the first parenthesized expression, and so on. To draw a parallel with the `replace()` method, `a[n]` holds the contents of `$n`.

For example, consider parsing a URL with the following code:

```
var url = /(\w+):\/\/\(([\\w.]+)\)\.(\S*)/;
var text = "Visit my home page at http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0]; // Contains "http://www.isp.com/~david"
    var protocol = result[1]; // Contains "http"
    var host = result[2]; // Contains "www.isp.com"
    var path = result[3]; // Contains "~david"
}
```

Finally, there is one more feature of the `match()` method that you should know about. The array it returns has a `length` property, as all arrays do. When `match()` is invoked on a nonglobal regular expression, however, the returned array also has two other properties: the `index` property, which contains the character position within the string at which the match begins; and the `input` property, which is a copy of the target string. So in the previous code, the value of the `result.index` property would be 21, since the matched URL begins at character position 21 in the text. The `result.input` property would hold the same string as the `text` variable. For a regular expression `r` that does not have the `g` flag set, calling `s.match(r)` returns the same value as `r.exec(s)`. We'll discuss the `RegExp.exec()` method a little later in this chapter.

The last of the regular expression methods of the `String` object is `split()`. This method breaks the string on which it is called into an array of substrings, using the argument as a separator. For example:

```
"123,456,789".split(","); // Returns ["123","456","789"]
```

The `split()` method can also take a regular expression as its argument. This ability makes the method more powerful. For example, we can now specify a separator character that allows an arbitrary amount of whitespace on either side:

```
"1,2, 3 , 4 ,5".split(/\s*,\s*/); // Returns ["1","2","3","4","5"]
```

The `split()` method has other features as well. See the "`String.split()`" entry in the core reference section for complete details.

10.3 The RegExp Object

As mentioned at the beginning of this chapter, regular expressions are represented as `RegExp` objects. In addition to the `RegExp()` constructor, `RegExp` objects support three methods and a number of properties. An unusual feature of the `RegExp` class is that it defines both class (or static) properties and instance properties. That is, it defines global properties that belong to the `RegExp()` constructor as well as other properties that belong to individual `RegExp` objects. `RegExp` pattern-matching methods and properties are described in the next two sections.

The `RegExp()` constructor takes one or two string arguments and creates a new `RegExp` object. The first argument to this constructor is a string that contains the body of the regular expression -- the text that would appear within slashes in a regular expression literal. Note that both string literals and regular expressions use the `\` character for escape sequences, so when you pass a regular expression to `RegExp()` as a string literal, you must replace each `\` character with `\\`. The second argument to `RegExp()` is optional. If supplied, it indicates the regular expression flags. It should be `g`, `i`, `m`, or a combination of those letters. For example:

```
// Find all five digit numbers in a string. Note the double \\ in this
case.
var zipcode = new RegExp("\\d{5}", "g");
```

The `RegExp()` constructor is useful when a regular expression is being dynamically created and thus cannot be represented with the regular expression literal syntax. For example, to search for a string entered by the user, a regular expression must be created at runtime with `RegExp()`.

10.3.1 RegExp Methods for Pattern Matching

RegExp objects define two methods that perform pattern-matching operations; they behave similarly to the String methods described earlier. The main RegExp pattern-matching method is `exec()`. It is similar to the String `match()` method described above, except that it is a RegExp method that takes a string, rather than a String method that takes a RegExp. The `exec()` method executes a regular expression on the specified string. That is, it searches the string for a match. If it finds none, it returns `null`. If it does find one, however, it returns an array just like the array returned by the `match()` method for nonglobal searches. Element 0 of the array contains the string that matched the regular expression, and any subsequent array elements contain the substrings that matched any parenthesized subexpressions. Furthermore, the `index` property contains the character position at which the match occurred, and the `input` property refers to the string that was searched.

Unlike the `match()` method, `exec()` returns the same kind of array whether or not the regular expression has the global `g` flag. Recall that `match()` returns an array of matches when passed a global regular expression. `exec()`, by contrast, always returns a single match and provides complete information about that match. When `exec()` is called for a regular expression that has the `g` flag, it sets the `lastIndex` property of the regular expression object to the character position immediately following the matched substring. When `exec()` is invoked a second time for the same regular expression, it begins its search at the character position indicated by the `lastIndex` property. If `exec()` does not find a match, it resets `lastIndex` to 0. (You can also set `lastIndex` to 0 at any time, which you should do whenever you quit a search before you find the last match in one string and begin searching another string with the same RegExp object.) This special behavior allows us to call `exec()` repeatedly in order to loop through all the regular expression matches in a string. For example:

```
var pattern = /Java/g;
var text = "JavaScript is more fun than Java!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched `" + result[0] + "`" +
        " at position " + result.index +
        "; next search begins at " + pattern.lastIndex);
}
```


The other RegExp method is `test()`. `test()` is a much simpler method than `exec()`. It takes a string and returns `true` if the string matches the regular expression:

```
var pattern = /java/i;
pattern.test("JavaScript"); // Returns true
```

Calling `test()` is equivalent to calling `exec()` and returning `true` if the return value of `exec()` is not `null`. Because of this equivalence, the `test()` method behaves the same way as the `exec()` method when invoked for a global regular expression: it begins searching the specified string at the position specified by `lastIndex`, and if it finds a match, it sets `lastIndex` to the position of the character immediately following the match. Thus, we can loop through a string using the `test()` method just as we can with the `exec()` method.

The String methods `search()`, `replace()`, and `match()` do not use the `lastIndex` property as `exec()` and `test()` do. In fact, the String methods simply reset `lastIndex()` to 0. If you use `exec()` or `test()` on a pattern that has the `g` flag set and you are searching multiple strings, you must either find all the matches in each string, so that `lastIndex` is automatically reset to zero (this happens when the last search fails), or you must explicitly set the `lastIndex` property to 0 yourself. If you forget to do this, you may start searching a new string at some arbitrary position within the string rather than from the beginning. Finally, remember that this special `lastIndex` behavior occurs only for regular expressions with the `g` flag. `exec()` and `test()` ignore the `lastIndex` property of RegExp objects that do not have the `g` flag.

10.3.2 RegExp Instance Properties

Each RegExp object has five properties. The `source` property is a read-only string that contains the text of the regular expression. The `global` property is a read-only boolean value that specifies whether the regular expression has the `g` flag. The `ignoreCase` property is a read-only boolean value that specifies whether the regular expression has the `i` flag. The `multiline` property is a read-only boolean value that specifies whether the regular expression has the `m` flag. The final property is `lastIndex`, a read-write integer. For patterns with the `g` flag, this property stores the position in the string at which the next search is to begin. It is used by the `exec()` and `test()` methods, as described in the previous section.

Chapter 11. Further Topics in JavaScript

This chapter covers miscellaneous JavaScript topics that would have bogged down previous chapters had they been covered earlier. Now that you have read through the preceding chapters and are experienced with the core JavaScript language, you are prepared to tackle the more advanced and detailed concepts presented here. You may prefer, however, to move on to other chapters and learn about the specifics of client-side JavaScript before returning to this chapter.

11.1 Data Type Conversion

We've seen that JavaScript is an untyped language (or, perhaps more accurately, a loosely typed or dynamically typed language). This means, for example, that we don't have to specify the data type of a variable when we declare it. Being untyped gives JavaScript the flexibility and simplicity that are desirable for a scripting language (although those features come at the expense of rigor, which is important for the longer, more complex programs often written in stricter languages such as C and Java). An important feature of JavaScript's flexible treatment of data types is the automatic type conversions it performs. For example, if you pass a number to the `document.write()` method, JavaScript automatically converts that value into its equivalent string representation. Similarly, if you test a string value in the condition of an `if` statement, JavaScript automatically converts that string to a boolean value -- to `false` if the string is empty and to `true` otherwise.

The basic rule is that when a value of one type is used in a context that requires a value of some other type, JavaScript automatically attempts to convert the value as needed. So, for example, if a number is used in a boolean context, it is converted to a boolean. If an object is used in a string context, it is converted to a string. If a string is used in a numeric context, JavaScript attempts to convert it to a number. [Table 11-1](#) summarizes each of these conversions -- it shows the conversion that is performed when a particular type of value is used in a particular context. The sections that follow the table provide more detail about type conversions in JavaScript.

Value	Context in which value is used			
	String	Number	Boolean	Object
Undefined value	"undefined"	NaN	false	Error
null	"null"	0	false	Error

Table 11-1. Automatic data type conversions

Value	Context in which value is used			
	String	Number	Boolean	Object
Nonempty string	As is	Numeric value of string or NaN	true	String object
Empty string	As is	0	false	String object
0	"0"	As is	false	Number object
NaN	"NaN"	As is	false	Number object
Infinity	"Infinity"	As is	true	Number object
Negative infinity	"-Infinity"	As is	true	Number object
Any other number	String value of number	As is	true	Number object
true	"true"	1	As is	Boolean object
false	"false"	0	As is	Boolean object
Object	toString()	valueOf() or toString() or NaN	true	As is

11.1.1 Object-to-Primitive Conversion

[Table 11-1](#) specifies how JavaScript objects are converted to primitive values. Several details of this conversion require additional discussion, however. First, note that whenever a non-null object is used in a boolean context, it converts to `true`. This is true for all objects (including all arrays and functions), even wrapper objects that represent primitive values that convert to `false`. For example, all of the following objects convert to `true` when used in a boolean context:^[1]

^[1] Note, though, that in JavaScript 1.1 and 1.2 these objects all convert to `false`, which is ECMAScript compliant.

```
new Boolean(false) // Internal value is false, but object converts to true
new Number(0)
```

```
new String("")
new Array( )
```

[Table 11-1](#) shows that objects are converted to numbers by first calling the `valueOf()` method of the object. Most objects inherit the default `valueOf()` method of `Object`, which simply returns the object itself. Since the default `valueOf()` method does not return a primitive value, JavaScript next tries to convert the object to a number by calling its `toString()` method and converting the resulting string to a number.

This leads to interesting results for arrays. Recall that the `toString()` method of arrays converts the array elements to strings, then returns the result of concatenating these strings, with commas in between. Therefore, an array with no elements converts to the empty string, which (as you can see in the table) converts to the number zero! Also, if an array has a single element that is a number *n*, the array converts to a string representation of *n*, which is then converted back to *n* itself. If an array contains more than one element, or if its one element is not a number, the array converts to `NaN`.^[2]

^[2] Note, however, that in JavaScript 1.1 and 1.2, when an array is used in a numeric context it is converted to its length.

[Table 11-1](#) specifies how an object is converted when used in a string context and how it is converted when used in a numeric context. However, there are a couple of places in JavaScript where the context is ambiguous! The `+` operator and the comparison operators (`<`, `<=`, `>`, and `>=`) operate on both numbers and strings, so when an object is used with one of these operators, it is not clear whether it should be converted to a number or a string. In most cases, JavaScript first attempts to convert the object by calling its `valueOf()` method. If this method returns a primitive value (usually a number), that value is used. Often, however, `valueOf()` simply returns the unconverted object; in this case, JavaScript then tries to convert the object to a string by calling its `toString()` method.

There is only one exception to this conversion rule: when a `Date` object is used with the `+` operator, conversion is performed with the `toString()` method. This exception exists because `Date` has both `toString()` and `valueOf()` methods. When a `Date` is used with `+`, you almost always want to perform a string concatenation. But when using a `Date` with the comparison operators, you almost always want to perform a numeric comparison to determine which of two times is earlier than the other.

Most objects either don't have `valueOf()` methods or don't have `valueOf()` methods that return useful results. When you use an object with the `+` operator, you usually get string concatenation rather than addition. When you use an object with a comparison operator, you usually get string comparison rather than numeric comparison.

An object that defines a custom `valueOf()` method may behave differently. If you define a `valueOf()` method that returns a number, you can use arithmetic and other operators with your object, but adding your object to a string may not behave as you

expect: the `toString()` method is no longer called, and a string representation of the number returned by `valueOf()` is concatenated to the string.

Finally, remember that `valueOf()` is not called `toNumber()`: strictly speaking, its job is to convert an object to a reasonable primitive value, so some objects may have `valueOf()` methods that return strings.

11.1.2 Explicit Type Conversions

[Table 11-1](#) listed the automatic data type conversions that JavaScript performs. It is also possible to explicitly convert values from one type to another. JavaScript does not define a cast operator as C, C++, and Java do, but it does provide similar facilities for converting data values.

As of JavaScript 1.1 (and the ECMA-262 standard), `Number()`, `Boolean()`, `String()`, and `Object()` may be called as functions as well as being invoked as constructors. When invoked in this way, these functions attempt to convert their arguments to the appropriate type. For example, you could convert any value `x` to a string with `String(x)` and convert any value `y` to an object with `Object(y)`.

There are a few other tricks that can be useful for performing explicit conversions. To convert a value to a string, concatenate it with the empty string:

```
var x_as_string = x + "";
```

To force a value to a number, subtract zero from it:

```
var x_as_number = x - 0;
```

And to force a value to boolean, use the `!` operator twice:

```
var x_as_boolean = !!x;
```

Because of JavaScript's tendency to automatically convert data to whatever type is required, explicit conversions are usually unnecessary. They are occasionally helpful, however, and can also be used to make your code clearer and more precise.

11.1.3 Converting Numbers to Strings

The number-to-string conversion is probably the one most often performed in JavaScript. Although it usually happens automatically, there are a couple of useful ways to explicitly perform this conversion. We saw two of them above:

```
var string_value = String(number); // Use the String( ) constructor
as a function
var string_value = number + ""; // Concatenate with the empty string
```

Another technique for converting numbers to strings is with the `toString()` method:

```
string_value = number.toString( );
```

The `toString()` method of the `Number` object (primitive numbers are converted to `Number` objects so that this method can be called) takes an optional argument that specifies a radix, or base, for the conversion. If you do not specify the argument, the conversion is done in base 10. But you can also convert numbers in other bases (between 2 and 36).^[3] For example:

^[3] Note that the ECMAScript specification supports the radix argument to the `toString()` method, but it allows the method to return an implementation-defined string for any radix other than 10. Thus, conforming implementations may simply ignore the argument and always return a base-10 result. In practice, implementations from Netscape and Microsoft do honor the requested radix.

```
var n = 17;
binary_string = n.toString(2); // Evaluates to "10001"
octal_string = "0" + n.toString(8); // Evaluates to "021"
hex_string = "0x" + n.toString(16); // Evaluates to "0x11"
```

A shortcoming of JavaScript prior to JavaScript 1.5 is that there is no built-in way to convert a number to a string and specify the number of decimal places to be included, or to specify whether exponential notation should be used. This can make it difficult to display numbers that have traditional formats, such as numbers that represent monetary values.

ECMAScript v3 and JavaScript 1.5 solve this problem by adding three new number-to-string methods to the `Number` class. `toFixed()` converts a number to a string and displays a specified number of digits after the decimal point. It does not use exponential notation. `toExponential()` converts a number to a string using exponential notation, with one digit before the decimal point and a specified number of digits after the decimal point. `toPrecision()` displays a number using the specified number of significant digits. It uses exponential notation if the number of significant digits is not large enough to display the entire integer portion of the number. Note that all three of these methods round the trailing digits of the resulting string as appropriate. Consider the following examples:

```
var n = 123456.789;
n.toFixed(0); // "123457"
n.toFixed(2); // "123456.79"
n.toExponential(1); // "1.2e+5"
n.toExponential(3); // "1.235e+5"
n.toPrecision(4); // "1.235e+5"
n.toPrecision(7); // "123456.8"
```

11.1.4 Converting Strings to Numbers

We've seen that strings that represent numbers are automatically converted to actual numbers when used in a numeric context. As shown earlier, we can make this conversion explicit:

```
var number = Number(string_value);
var number = string_value - 0;
```

The trouble with this sort of string-to-number conversion is that it is overly strict. It works only with base-10 numbers, and although it does allow leading and trailing spaces, it does not allow any non-space characters to appear in the string following the number.

To allow more flexible conversions, you can use `parseInt()` and `parseFloat()`. These functions convert and return any number at the beginning of a string, ignoring any trailing non-numbers. `parseInt()` parses only integers, while `parseFloat()` parses both integers and floating-point numbers. If a string begins with "0x" or "0X", `parseInt()` interprets it as a hexadecimal number.^[4] For example:

^[4] The ECMAScript specification says that if a string begins with "0" (but not "0x" or "0X"), `parseInt()` may parse it as an octal number or as a decimal number. Because the behavior is unspecified, you should never use `parseInt()` to parse numbers with leading zeros, unless you explicitly specify the radix to be used!

```
parseInt("3 blind mice");      // Returns 3
parseFloat("3.14 meters");    // Returns 3.14
parseInt("12.34");            // Returns 12
parseInt("0xFF");             // Returns 255
```

`parseInt()` can even take a second argument specifying the radix (base) of the number to be parsed. Legal values are between 2 and 36. For example:

```
parseInt("11", 2);             // Returns 3 (1*2 + 1)
parseInt("ff", 16);           // Returns 255 (15*16 + 15)
parseInt("zz", 36);           // Returns 1295 (35*36 + 35)
parseInt("077", 8);           // Returns 63 (7*8 + 7)
parseInt("077", 10);          // Returns 77 (7*10 + 7)
```

If `parseInt()` or `parseFloat()` cannot convert the specified string to a number, it returns `NaN`:

```
parseInt("eleven");           // Returns NaN
parseFloat("$72.47");         // Returns NaN
```

11.2 By Value Versus by Reference

In JavaScript, as in all programming languages, there are three important ways that you can manipulate a data value. First, you can copy it; for example, by assigning it to a new variable. Second, you can pass it as an argument to a function or method. Third, you can compare it with another value to see if the two values are equal. To understand any programming language, you must understand how these three operations are performed in that language.

There are two fundamentally distinct ways to manipulate data values. These techniques are called "by value" and "by reference." When a value is manipulated by value, it is the *value* of the datum that matters. In an assignment, a copy of the actual value is made and that copy is stored in a variable, object property, or array element; the copy and the original are two totally independent values that are stored separately. When a datum is passed by value to a function, a copy of the datum is passed to the function; if the function modifies the value, the change affects only the function's copy of the datum -- it does not affect the original datum. Finally, when a datum is compared by value to another datum, the two distinct pieces of data must represent exactly the same value (which usually means that a byte-by-byte comparison finds them to be equal).

The other way of manipulating a value is by reference. With this technique, there is only one actual copy of the value; references to that value are manipulated.^[5] If a value is manipulated by reference, variables do not hold that value directly; they hold only references to it. It is these references that are copied, passed, and compared. So, in an assignment made by reference, it is the reference to the value that is assigned, not a copy of the value and not the value itself. After the assignment, the new variable refers to the same value that the original variable refers to. Both references are equally valid and both can be used to manipulate the value -- if the value is changed through one reference, that change also appears through the original reference. The situation is similar when a value is passed to a function by reference. A reference to the value is passed to the function, and the function can use that reference to modify the value itself; any such modifications are visible outside the function. Finally, when a value is compared to another by reference, the two references are compared to see if they refer to the same unique copy of a value; references to two distinct values that happen to be equivalent (i.e., consist of the same bytes) are not treated as equal.

^[5] C programmers and anyone else familiar with the concept of pointers should understand the idea of a reference in this context. Note, however, that JavaScript does not support pointers.

These are two very different ways of manipulating values, and they have important implications that you should understand. [Table 11-2](#) summarizes these implications. This discussion of manipulating data by value and by reference has been a general one, but the distinctions apply to all programming languages. The sections that follow explain how these distinctions apply specifically to JavaScript; they discuss which data types are manipulated by value and which are manipulated by reference.

Table 11-2. By value versus by reference

	By value	By reference
Copy	The value is actually copied; there are two distinct, independent copies.	Only a reference to the value is copied. If the value is modified through the new reference, that change is also visible through the original reference.
Pass	A distinct copy of the value is passed to the function; changes to it have no effect outside the function.	A reference to the value is passed to the function. If the function modifies the value through the passed reference, the modification is visible outside the function.
Compare	Two distinct values are compared (often byte by byte) to see if they are the same value.	Two references are compared to see if they refer to the same value. Two references to distinct values are not equal, even if the two values consist of the same bytes.

11.2.1 Primitive Types and Reference Types

The basic rule in JavaScript is this: primitive types are manipulated by value, and reference types, as the name suggests, are manipulated by reference. Numbers and booleans are primitive types in JavaScript -- primitive because they consist of nothing more than a small, fixed number of bytes that are easily manipulated at the low (primitive) levels of the JavaScript interpreter. Objects, on the other hand, are reference types. Arrays and functions, which are specialized types of objects, are therefore also reference types. These data types can contain arbitrary numbers of properties or elements, so they cannot be manipulated as easily as fixed-size primitive values can. Since object and array values can become quite large, it doesn't make sense to manipulate these types by value, as this could involve the inefficient copying and comparing of large amounts of memory.

What about strings? A string can have an arbitrary length, so it would seem that strings should be reference types. In fact, though, they are usually considered to be primitive types in JavaScript simply because they are not objects. Strings don't actually fit into the primitive versus reference type dichotomy. We'll have more to say about strings and their behavior a little later.

The best way to explore the differences between data manipulation by value and by reference is through example. Study the following examples carefully, paying attention to the comments. [Example 11-1](#) copies, passes, and compares numbers. Since numbers are primitive types, this example illustrates data manipulation by value.

Example 11-1. Copying, passing, and comparing by value

```
// First we illustrate copying by value
```

```

var n = 1; // Variable n holds the value 1
var m = n; // Copy by value: variable m holds a distinct value 1

// Here's a function we'll use to illustrate passing by value
// As we'll see, the function doesn't work the way we'd like it to
function add_to_total(total, x)
{
    total = total + x; // This line changes only the internal copy of
total
}

// Now call the function, passing the numbers contained in n and m by
value.
// The value of n is copied, and that copied value is named total
within the
// function. The function adds a copy of m to that copy of n. But
adding
// something to a copy of n doesn't affect the original value of n
outside
// of the function. So calling this function doesn't accomplish
anything.
add_to_total(n, m);

// Now, we'll look at comparison by value.
// In the following line of code, the literal 1 is clearly a distinct
numeric
// value encoded in the program. We compare it to the value held in
variable
// n. In comparison by value, the bytes of the two numbers are checked
to
// see if they are the same.
if (n == 1) m = 2; // n contains the same value as the literal 1; m is
now 2

```

Now, consider [Example 11-2](#). This example copies, passes, and compares an object. Since objects are reference types, these manipulations are performed by reference. This example uses Date objects, which you can read more about in the core reference section of this book.

Example 11-2. Copying, passing, and comparing by reference

```

// Here we create an object representing the date of Christmas, 2001
// The variable xmas contains a reference to the object, not the object
itself
var xmas = new Date(2001, 11, 25);

// When we copy by reference, we get a new reference to the original
object
var solstice = xmas; // Both variables now refer to the same object
value

// Here we change the object through our new reference to it
solstice.setDate(21);

// The change is visible through the original reference, as well

```

```

xmas.getDate( ); // Returns 21, not the original value of 25

// The same is true when objects and arrays are passed to functions.
// The following function adds a value to each element of an array.
// A reference to the array is passed to the function, not a copy of
the array.
// Therefore, the function can change the contents of the array through
// the reference, and those changes will be visible when the function
returns.
function add_to_totals(totals, x)
{
    totals[0] = totals[0] + x;
    totals[1] = totals[1] + x;
    totals[2] = totals[2] + x;
}

// Finally, we'll examine comparison by reference.
// When we compare the two variables defined above, we find they are
// equal, because they refer to the same object, even though we were
trying
// to make them refer to different dates:
(xmas == solstice) // Evaluates to true

// The two variables defined next refer to two distinct objects, both
// of which represent exactly the same date.
var xmas = new Date(2001, 11, 25);
var solstice_plus_4 = new Date(2001, 11, 25);

// But, by the rules of "compare by reference," distinct objects are
not equal!
(xmas != solstice_plus_4) // Evaluates to true

```

Before we leave the topic of manipulating objects and arrays by reference, we need to clear up a point of nomenclature. The phrase "pass by reference" can have several meanings. To some readers, the phrase refers to a function invocation technique that allows a function to assign new values to its arguments and to have those modified values visible outside the function. This is not the way the term is used in this book. Here, we mean simply that a reference to an object or array -- not the object itself -- is passed to a function. A function can use the reference to modify properties of the object or elements of the array. But if the function overwrites the reference with a reference to a new object or array, that modification is not visible outside of the function. Readers familiar with the other meaning of this term may prefer to say that objects and arrays are passed by value, but the value that is passed is actually a reference rather than the object itself. [Example 11-3](#) illustrates this issue.

Example 11-3. References themselves are passed by value

```

// This is another version of the add_to_totals( ) function. It
doesn't
// work, though, because instead of changing the array itself, it tries
to
// change the reference to the array.
function add_to_totals2(totals, x)

```

```
{
  newtotals = new Array(3);
  newtotals[0] = totals[0] + x;
  newtotals[1] = totals[1] + x;
  newtotals[2] = totals[2] + x;
  totals = newtotals; // This line has no effect outside of the
function
}
```

11.2.2 Copying and Passing Strings

As mentioned earlier, JavaScript strings don't fit neatly into the primitive type versus reference type dichotomy. Since strings are not objects, it is natural to assume that they are primitive. If they are primitive types, then by the rules given above, they should be manipulated by value. But since strings can be arbitrarily long, it would seem inefficient to copy, pass, and compare them byte by byte. Therefore, it would also be natural to assume that strings are implemented as reference types.

Instead of making assumptions about strings, suppose we write some JavaScript code to experiment with string manipulation. If strings are copied and passed by reference, we should be able to modify the contents of a string through the reference stored in another variable or passed to a function.

When we set out to write the code to perform this experiment, however, we run into a major stumbling block: there is no way to modify the contents of a string. The `charAt()` method returns the character at a given position in a string, but there is no corresponding `setCharAt()` method. This is not an oversight. JavaScript strings are intentionally *immutable* -- that is, there is no JavaScript syntax, method, or property that allows you to change the characters in a string.

Since strings are immutable, our original question is moot: there is no way to tell if strings are passed by value or by reference. We can assume that, for efficiency, JavaScript is implemented so that strings are passed by reference, but in actuality it doesn't matter, since it has no practical bearing on the code we write.

11.2.3 Comparing Strings

Despite the fact that we cannot determine whether strings are copied and passed by value or by reference, we can write JavaScript code to determine whether they are compared by value or by reference. [Example 11-4](#) shows the code we might use to make this determination.

Example 11-4. Are strings compared by value or by reference?

```
// Determining whether strings are compared by value or reference is
easy.
```

```

// We compare two clearly distinct strings that happen to contain the
// same
// characters. If they are compared by value they will be equal, but if
// they
// are compared by reference, they will not be equal:
var s1 = "hello";
var s2 = "hell" + "o";
if (s1 == s2) document.write("Strings compared by value");

```

This experiment demonstrates that strings are compared by value. This may be surprising to some programmers. In C, C++, and Java, strings are reference types and are compared by reference. If you want to compare the actual contents of two strings, you must use a special method or function. JavaScript, however, is a higher-level language and recognizes that when you compare strings, you most often want to compare them by value. Thus, despite the fact that, for efficiency, JavaScript strings are (presumably) copied and passed by reference, they are compared by value.

11.2.4 By Value Versus by Reference: Summary

[Table 11-3](#) summarizes the way that the various JavaScript types are manipulated.

<i>Table 11-3. Data type manipulation in JavaScript</i>			
Type	Copied by	Passed by	Compared by
Number	Value	Value	Value
Boolean	Value	Value	Value
String	Immutable	Immutable	Value
Object	Reference	Reference	Reference

11.3 Garbage Collection

As explained in [Chapter 4](#), JavaScript uses garbage collection to reclaim the memory occupied by strings, objects, arrays, and functions that are no longer in use. This frees you, the programmer, from having to explicitly deallocate memory yourself and is an important part of what makes JavaScript programming easier than, say, C programming.

A key feature of garbage collection is that the garbage collector must be able to determine when it is safe to reclaim memory. Obviously, it must never reclaim values that are still in use and should collect only values that are no longer reachable; that is, values that cannot be referred to through any of the variables, object properties, or array elements in the program. If you are the curious type, you may be wondering just how a garbage collector distinguishes between garbage to be collected and values that are still

being used or that could potentially be used. The following sections explain some of the gory details.

11.3.1 Mark-and-Sweep Garbage Collection

The computer science literature on garbage collection is large and technical; the actual operation of the garbage collector is really an implementation-specific detail that may vary in different implementations of the language. Still, almost all serious garbage collectors use some variation on a basic garbage-collection algorithm known as "mark and sweep."

A mark-and-sweep garbage collector periodically traverses the list of all variables in the JavaScript environment and marks any values referred to by these variables. If any referenced values are objects or arrays, it recursively marks the object properties and array elements. By recursively traversing this tree or graph of values, the garbage collector is able to find (and mark) every single value that is still reachable. It follows, then, that any unmarked values are unreachable and are therefore garbage.

Once a mark-and-sweep garbage collector has finished marking all reachable values, it begins its sweep phase. During this phase, it looks through the list of all values in the environment and deallocates any that are not marked. Classic mark-and-sweep garbage collectors do a complete mark and a complete sweep all at once, which causes a noticeable slowdown in the system during garbage collection. More sophisticated variations on the algorithm make the process relatively efficient and perform collection in the background, without disrupting system performance.

The details of garbage collection are implementation-specific, and you should not need to know anything about the garbage collector to write JavaScript programs. All modern JavaScript implementations use some kind of mark-and-sweep garbage collection. However, JavaScript 1.1, as implemented in Netscape 3, used a somewhat simpler garbage-collection scheme that has some shortcomings. If you are writing JavaScript code to be compatible with Netscape 3, the following section explains the shortcomings of the garbage collector in that browser. Netscape 2 used an even simpler garbage-collection technique with serious flaws. Since that browser is now entirely obsolete, the details are not described here.

11.3.2 Garbage Collection by Reference Counting

In JavaScript 1.1, as implemented in Netscape 3, garbage collection is performed by reference counting. This means that every object (whether a user object created by JavaScript code or a built-in HTML object created by the browser) keeps track of the number of references to it. Recall that objects are assigned by reference in JavaScript, rather than having their complete values copied.

When an object is created and a reference to it is stored in a variable, the object's reference count is one. When the reference to the object is copied and stored in another

variable, the reference count is incremented to two. When one of the two variables that holds these references is overwritten with some new value, the object's reference count is decremented back to one. If the reference count reaches zero, there are no more references to the object. Since there are no references to copy, there can never again be a reference to the object in the program. Therefore, JavaScript knows that it is safe to destroy the object and garbage collect the memory associated with it.

Unfortunately, there are shortcomings to using reference counting as a garbage-collection scheme. In fact, some people don't even consider reference counting to be true garbage collection and reserve that term for better algorithms, such as mark-and-sweep garbage collection. Reference counting is a simple form of garbage collection that is easy to implement and works fine in many situations. There is an important situation, however, in which reference counting cannot correctly detect and collect all garbage, and you need to be aware of it.

The basic flaw with reference counting has to do with *cyclical references*. If object A contains a reference to object B and object B contains a reference to object A, a cycle of references exists. A cycle would also exist, for example, if A referred to B, B referred to C, and C referred back to A. In cycles such as these, there is always a reference from within the cycle to every element in the cycle. Thus, even if none of the elements of the cycle has any remaining outside references, their reference counts will never drop below one and they can never be garbage collected. The entire cycle may be garbage if there is no way to refer to any of these objects from a program, but because they all refer to each other, a reference-counting garbage collector cannot detect and free this unused memory.

This problem with cycles is the price that must be paid for a simple garbage-collection scheme. The only way to prevent this problem is by manual intervention. If you create a cycle of objects, you must recognize this fact and take steps to ensure that the objects are garbage collected when they are no longer needed. To allow a cycle of objects to be garbage collected, you must break the cycle. You can do this by picking one of the objects in the cycle and setting the property of it that refers to the next object to `null`. For example, suppose that A, B, and C are objects that each have a `next` property, and the value of this property is set so that these objects refer to each other and form a cycle. When these objects are no longer in use, you can break the cycle by setting `A.next` to `null`. This means that object B no longer has a reference from A, so its reference count can drop to zero and it can be garbage collected. Once it has been garbage collected, it will no longer refer to C, so C's reference count can drop to zero and it can be garbage collected. Once C is garbage collected, A can finally be garbage collected.

Note, of course, that none of this can happen if A, B, and C are stored in global variables in a window that is still open, because those variables A, B, and C still refer to the objects. If these were local variables in a function and you broke their cycle before the function returned, they could be garbage collected. But if they are stored in global variables, they remain referenced until the window that contains them closes. In this case, if you want to force them to be garbage collected, you must break the cycle and set all the variables to `null`:

```
A.next = null;      // Break the cycle
A = B = C = null;  // Remove the last remaining external references
```

11.4 Lexical Scoping and Nested Functions

Functions in JavaScript are lexically rather than dynamically scoped. This means that they run in the scope in which they are defined, not the scope from which they are executed. Prior to JavaScript 1.2, functions could be defined only in the global scope, and lexical scoping was not much of an issue: all functions were executed in the same global scope (with the call object of the function chained to that global scope).

In JavaScript 1.2 and later, however, functions can be defined anywhere, and tricky issues of scope arise. For example, consider a function `g` defined within a function `f`. `g` is always executed in the scope of `f`. Its scope chain includes three objects: its own call object, the call object of `f()`, and the global object. Nested functions are perfectly understandable when they are invoked in the same lexical scope in which they are defined. For example, the following code does not do anything particularly surprising:

```
var x = "global";
function f( ) {
    var x = "local";
    function g( ) { alert(x); }
    g( );
}
f( ); // Calling this function displays "local"
```

In JavaScript, however, functions are data just like any other value, so they can be returned from functions, assigned to object properties, stored in arrays, and so on. This does not cause anything particularly surprising either, except when nested functions are involved. Consider the following code, which includes a function that returns a nested function. Each time it is called, it returns a function. The JavaScript code of the returned function is always the same, but the scope in which it is created differs slightly on each invocation, because the values of the arguments to the outer function differ on each invocation. If we save the returned functions in an array and then invoke each one, we'll see that each returns a different value. Since each function consists of identical JavaScript code and each is invoked from exactly the same scope, the only factor that could be causing the differing return values is the scope in which the functions were defined:

```
// This function returns a function each time it is called
// The scope in which the function is defined differs for each call
function makefunc(x) {
    return function( ) { return x; }
}

// Call makefunc( ) several times, and save the results in an array:
var a = [makefunc(0), makefunc(1), makefunc(2)];

// Now call these functions and display their values.
```



```
// Although the body of each function is the same, the scope is
// different, and each call returns a different value:
alert(a[0]( )); // Displays 0
alert(a[1]( )); // Displays 1
alert(a[2]( )); // Displays 2
```

The results of this code may be surprising. Still, they are the results expected from a strict application of the lexical scoping rule: a function is executed in the scope in which it was defined. That scope includes the state of local variables and arguments. Even though local variables and function arguments are transient, their state is frozen and becomes part of the lexical scope of any functions defined while they are in effect. In order to make lexical scoping work with nested functions, a JavaScript implementation must use a *closure*, which can be thought of as a combination of a function definition and the scope chain that was in effect when the function was defined.

11.5 The Function() Constructor and Function Literals

As we saw in [Chapter 7](#), there are two ways to define functions other than the basic `function` statement. As of JavaScript 1.1, functions can be defined using the `Function()` constructor, and in JavaScript 1.2 and later, they can be defined with function literals. You should be aware of some important differences between these two techniques.

First, the `Function()` constructor allows JavaScript code to be dynamically created and compiled at runtime. Function literals, however, are a static part of program structure, just as `function` statements are.

Second, as a corollary of the first difference, the `Function()` constructor parses the function body and creates a new function object each time it is called. If the call to the constructor appears within a loop or within a frequently called function, this process can be inefficient. On the other hand, a function literal or nested function that appears within a loop or function is not recompiled each time it is encountered. Nor is a different function object created each time a function literal is encountered. (Although, as noted earlier, a new closure may be required to capture differences in the lexical scope in which the function is defined.)

The third difference between the `Function()` constructor and function literals is that functions created with the `Function()` constructor do not use lexical scoping; instead, they are always compiled as if they were top-level functions, as the following code demonstrates:

```
var y = "global";
function constructFunction( ) {
    var y = "local";
    return new Function("return y"); // Does not capture the local
scope!
```

```
}  
// This line displays "global", because the function returned by the  
// Function( ) constructor does not use the local scope. Had a  
function  
// literal been used instead, this line would have displayed "local".  
alert(constructFunction()( )); // Displays "global"
```

11.6 Netscape's JavaScript 1.2 Incompatibilities

Netscape's implementation of JavaScript 1.2 was released (as part of the Netscape 4.0 browser) while the ECMAScript v1 specification was still being finalized. The engineers at Netscape made some guesses about what would be in the specification, and based on those guesses, they made some changes to the way JavaScript behaved. Because these changes were not compatible with previous versions of JavaScript, the changes were implemented only when JavaScript 1.2 was explicitly requested. (In web browsers, this is done by setting the `language` attribute of the HTML `<script>` tag to "JavaScript1.2".) This was an excellent way to introduce new behavior without breaking old scripts. Unfortunately, when work on ECMAScript v1 was completed, the new behavior that Netscape engineers had guessed at was not part of the standard. What this means is that Netscape's implementation of JavaScript 1.2 has special-case behavior that is not compatible with JavaScript 1.1 and does not conform to the ECMAScript specification.

For compatibility with scripts that rely on the nonconforming behavior of JavaScript 1.2, all future implementations of JavaScript from Netscape have retained this special behavior when Version 1.2 is explicitly requested. Note, however, that if you request a version greater than 1.2 (with a `language` attribute of "JavaScript1.3", for example) you will get ECMAScript-compliant behavior. Because this special behavior is present only in JavaScript implementations from Netscape, you should not rely on it in your scripts, and the best practice is to never explicitly specify Version 1.2. Nevertheless, for those cases when you must use JavaScript 1.2, the special behaviors of that version are listed here:

- The equality and inequality operators behave like the identity and non-identity operators. That is, `==` works like `===` and `!=` works like `!==`.
- The default `Object.toString()` method displays the values of all properties defined by the object, returning a string formatted using object literal syntax.
- The `Array.toString()` method separates array elements with a comma and a space, instead of just a comma, and returns the list of elements within square brackets. In addition, string elements of the array are quoted, so that the result is a string in legal array literal syntax.
- When a single numeric argument `n` is passed to the `Array()` constructor, it returns an array with `n` as its single element, rather than an array of length `n`.
- When an array object is used in a numeric context, it evaluates to its length. When used in a boolean context, it evaluates to `false` if its length is 0 and otherwise evaluates to `true`.
- The `Array.push()` method returns the last value pushed rather than the new array length.

- When the `Array.splice()` method splices out a single element `x`, it returns `x` itself, rather than an array containing `x` as its only element. When `splice()` does not remove any elements from the array, it returns nothing instead of returning an empty array.
- When `String.substring()` is called with a starting position greater than its ending position, it returns the empty string rather than correctly swapping the arguments and returning the substring between them.
- The `String.split()` method displays special behavior inherited from Perl: if the specified separator character is a single space, it discards any leading and trailing whitespace in the string before splitting the remainder of the string.

Part II: Client-Side JavaScript

This part of the book, [Chapter 12](#) through [Chapter 22](#), documents JavaScript as it is implemented in web browsers. These chapters introduce a host of new JavaScript objects that represent the web browser and the contents of HTML documents. Many examples show typical uses of these client-side objects. You will find it helpful to study these examples carefully.

- [Chapter 12](#)
- [Chapter 13](#)
- [Chapter 14](#)
- [Chapter 15](#)
- [Chapter 16](#)
- [Chapter 17](#)
- [Chapter 18](#)
- [Chapter 19](#)
- [Chapter 20](#)
- [Chapter 21](#)
- [Chapter 22](#)

Chapter 12. JavaScript in Web Browsers

The first part of this book described the core JavaScript language. Now we move on to JavaScript as used within web browsers, commonly called client-side JavaScript.^[1] Most of the examples we've seen so far, while legal JavaScript code, had no particular context; they were JavaScript fragments that ran in no specified environment. This chapter provides that context. It begins with a conceptual introduction to the web browser programming environment and basic client-side JavaScript concepts. Next, it discusses how we actually embed JavaScript code within HTML documents so it can run in a web browser. Finally, the chapter goes into detail about how JavaScript programs are executed in a web browser.

^[1] The term "client-side JavaScript" is left over from the days when JavaScript was used in only two places: web browsers (clients) and web servers. As JavaScript is adopted as a scripting language in more and more environments, the term client-side makes less and less sense because it doesn't specify the client side of *what*. Nevertheless, we'll continue to use the term in this book.

12.1 The Web Browser Environment

To understand client-side JavaScript, you must understand the conceptual framework of the programming environment provided by a web browser. The following sections introduce three important features of that programming environment:

- The Window object that serves as the global object and global execution context for client-side JavaScript code
- The client-side object hierarchy and the document object model that forms a part of it
- The event-driven programming model

12.1.1 The Window as Global Execution Context

The primary task of a web browser is to display HTML documents in a window. In client-side JavaScript, the Document object represents an HTML document, and the Window object represents the window (or frame) that displays the document. While the Document and Window objects are both important to client-side JavaScript, the Window object is more important, for one substantial reason: the Window object is the global object in client-side programming.

Recall from [Chapter 4](#) that in every implementation of JavaScript there is always a global object at the head of the scope chain; the properties of this global object are global variables. In client-side JavaScript, the Window object is the global object. The Window object defines a number of properties and methods that allow us to manipulate the web browser window. It also defines properties that refer to other important objects, such as the `document` property for the Document object. Finally, the Window object has two self-

referential properties, `window` and `self`. You can use either of these global variables to refer directly to the Window object.

Since the Window object is the global object in client-side JavaScript, all global variables are defined as properties of the window. For example, the following two lines of code perform essentially the same function:

```
var answer = 42;      // Declare and initialize a global variable
window.answer = 42;  // Create a new property of the Window object
```

The Window object represents a web browser window or a frame within a window. To client-side JavaScript, top-level windows and frames are essentially equivalent. It is common to write JavaScript applications that use multiple frames, and it is possible, if less common, to write applications that use multiple windows. Each window or frame involved in an application has a unique Window object and defines a unique execution context for client-side JavaScript code. In other words, a global variable declared by JavaScript code in one frame is not a global variable within a second frame. However, the second frame *can* access a global variable of the first frame; we'll see how when we consider these issues in more detail in [Chapter 13](#).

12.1.2 The Client-Side Object Hierarchy and the Document Object Model

We've seen that the Window object is the key object in client-side JavaScript. All other client-side objects are connected to this object. For example, every Window object contains a `document` property that refers to the Document object associated with the window and a `location` property that refers to the Location object associated with the window. A Window object also contains a `frames[]` array that refers to the Window objects that represent the frames of the original window. Thus, `document` represents the Document object of the current window, and `frames[1].document` refers to the Document object of the second child frame of the current window.

An object referenced through the current window or through some other Window object may itself refer to other objects. For example, every Document object has a `forms[]` array containing Form objects that represent any HTML forms appearing in the document. To refer to one of these forms, you might write:

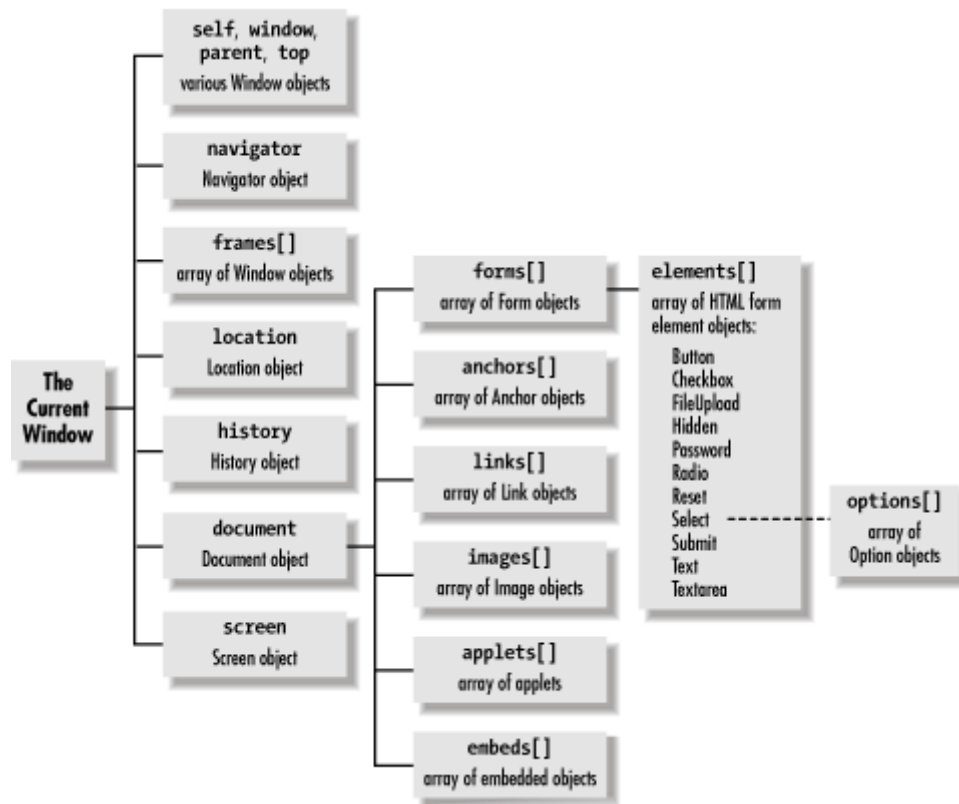
```
window.document.forms[0]
```

To continue with the same example, each Form object has an `elements[]` array containing objects that represent the various HTML form elements (input fields, buttons, etc.) that appear within the form. In extreme cases, you can write code that refers to an object at the end of a whole chain of objects, ending up with expressions as complex as this one:

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

We've seen that the Window object is the global object at the head of the scope chain and that all client-side objects in JavaScript are accessible as properties of other objects. This means that there is a hierarchy of JavaScript objects, with the Window object at its root. [Figure 12-1](#) shows this hierarchy. Study this figure carefully; understanding the hierarchy and the objects it contains is crucial to successful client-side JavaScript programming. Most of the remaining chapters of this book are devoted to fleshing out the details of the objects shown in this figure.

Figure 12-1. The client-side object hierarchy and Level 0 DOM



Note that [Figure 12-1](#) shows just the object properties that refer to other objects. Most of the objects shown in the diagram have quite a few more properties than those shown.

Many of the objects pictured in [Figure 12-1](#) descend from the Document object. This subtree of the larger client-side object hierarchy is known as the document object model (DOM), which is interesting because it has been the focus of a standardization effort. [Figure 12-1](#) illustrates the Document objects that have become de facto standards because they are consistently implemented by all major browsers. Collectively, they are known as the Level 0 DOM, because they form a base level of document functionality that JavaScript programmers can rely on in all browsers. These basic Document objects are

the subject of [Chapter 14](#) and [Chapter 15](#). A more advanced document object model that has been standardized by the W3C is the subject of [Chapter 17](#) and [Chapter 18](#).

12.1.3 The Event-Driven Programming Model

In the old days, computer programs often ran in batch mode -- they read in a batch of data, did some computation on that data, and then wrote out the results. Later, with time-sharing and text-based terminals, limited kinds of interactivity became possible -- the program could ask the user for input, and the user could type in data. The computer could then process the data and display the results on screen.

Nowadays, with graphical displays and pointing devices like mice, the situation is different. Programs are generally event driven; they respond to asynchronous user input in the form of mouse-clicks and keystrokes in a way that depends on the position of the mouse pointer. A web browser is just such a graphical environment. An HTML document contains an embedded graphical user interface (GUI), so client-side JavaScript uses the event-driven programming model.

It is perfectly possible to write a static JavaScript program that does not accept user input and does exactly the same thing every time. Sometimes this sort of program is useful. More often, however, we want to write dynamic programs that interact with the user. To do this, we must be able to respond to user input.

In client-side JavaScript, the web browser notifies programs of user input by generating *events*. There are various types of events, such as keystroke events, mouse motion events, and so on. When an event occurs, the web browser attempts to invoke an appropriate *event handler* function to respond to the event. Thus, to write dynamic, interactive client-side JavaScript programs, we must define appropriate event handlers and register them with the system, so that the browser can invoke them at appropriate times.

If you are not already accustomed to the event-driven programming model, it can take a little getting used to. In the old model, you wrote a single, monolithic block of code that followed some well-defined flow of control and ran to completion from beginning to end. Event-driven programming stands this model on its head. In event-driven programming, you write a number of independent (but mutually interacting) event handlers. You do not invoke these handlers directly, but allow the system to invoke them at the appropriate times. Since they are triggered by the user's input, the handlers will be invoked at unpredictable, asynchronous times. Much of the time, your program is not running at all but merely sitting waiting for the system to invoke one of its event handlers.

The next section explains how JavaScript code is embedded within HTML files. It shows how we can define both static blocks of code that run synchronously from start to finish and event handlers that are invoked asynchronously by the system. We'll also discuss events and event handling in much greater detail in [Chapter 19](#).

12.2 Embedding JavaScript in HTML

Client-side JavaScript code is embedded within HTML documents in a number of ways:

- Between a pair of `<script>` and `</script>` tags
- From an external file specified by the `src` attribute of a `<script>` tag
- In an event handler, specified as the value of an HTML attribute such as `onclick` or `onmouseover`
- As the body of a URL that uses the special `javascript:` protocol

The following sections document each of these JavaScript embedding techniques in more detail. Together, they explain all the ways to include JavaScript in web pages -- that is, they explain the allowed structure of JavaScript programs on the client side.

12.2.1 The `<script>` Tag

Client-side JavaScript scripts are part of an HTML file and are coded within `<script>` and `</script>` tags. You may place any number of JavaScript statements between these tags; they are executed in order of appearance, as part of the document loading process. `<script>` tags may appear in either the `<head>` or `<body>` of an HTML document.

A single HTML document may contain any number of nonoverlapping pairs of `<script>` and `</script>` tags. These multiple, separate scripts are executed in the order in which they appear within the document. While separate scripts within a single file are executed at different times during the loading and parsing of the HTML file, they constitute part of the same JavaScript program: functions and variables defined in one script are available to all scripts that follow in the same file. For example, you can have the following script somewhere in an HTML page:

```
<script>var x = 1;</script>
```

Later on in the same HTML page, you can refer to `x`, even though it's in a different script block. The context that matters is the HTML page, not the script block:

```
<script>document.write(x);</script>
```

The `document.write()` method is an important and commonly used one. When used as shown here, it inserts its output into the document at the location of the script. When the script finishes executing, the HTML parser resumes parsing the document and starts by parsing any text produced with `document.write()`.

[Example 12-1](#) shows a sample HTML file that includes a simple JavaScript program. Note the difference between this example and many of the code fragments shown earlier in this book: this one is integrated with an HTML file and has a clear context in which it runs. Note also the use of a `language` attribute in the `<script>` tag. This is explained in the next section.

Example 12-1. A simple JavaScript program in an HTML file

```
<html>
<head>
<title>Today's Date</title>
  <script language="JavaScript">
    // Define a function for later use
    function print_todays_date( ) {
      var d = new Date( ); // Get today's date and
time
      document.write(d.toLocaleString( )); // Insert it into the
document
    }
  </script>
</head>
<body>
The date and time are:<br>
<script language="JavaScript">
  // Now call the function we defined above
  print_todays_date( );
</script>
</body>
</html>
```

12.2.1.1 The language and type attributes

Although JavaScript is by far the most commonly used client-side scripting language, it is not the only one. In order to tell a web browser what language a script is written in, the `<script>` tag has an optional `language` attribute. Browsers that understand the specified scripting language run the script; browsers that do not know the language ignore it.

If you are writing JavaScript code, use the `language` attribute as follows:

```
<script language="JavaScript">
  // JavaScript code goes here
</script>
```

If, for example, you are writing a script in Microsoft's Visual Basic Scripting Edition language,^[2] you would use the attribute like this:

^[2] Also known as VBScript. The only browser that supports VBScript is Internet Explorer, so scripts written in this language are not portable. VBScript interfaces with HTML objects in the same way that JavaScript does, but the core language itself has a different syntax than JavaScript. VBScript is not documented in this book.

```
<script language="VBScript">
  ' VBScript code goes here ( ' is a comment character like // in
JavaScript)
</script>
```

JavaScript is the default scripting language for the Web, and if you omit the `language` attribute, both Netscape and Internet Explorer will assume that your scripts are written in JavaScript.

The HTML 4 specification standardizes the `<script>` tag, but it deprecates the `language` attribute because there is no standard set of names for scripting languages. Instead, the specification prefers the use of a `type` attribute that specifies the scripting language as a MIME type. Thus, in theory, the preferred way to embed a JavaScript script is with a tag that looks like this:

```
<script type="text/javascript">
```

In practice, the `language` attribute is still better supported than this new `type` attribute.

The HTML 4 specification also defines a standard (and useful) way to specify the default scripting language for an entire HTML file. If you plan to use JavaScript as the only scripting language in a file, simply include the following line in the `<head>` of the document:

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

If you do this, you can safely use JavaScript scripts without specifying the `language` or `type` attributes.

Since JavaScript is the default scripting language, those of us who program with it never really need to use the `language` attribute to specify the language in which a script is written. However, there is an important secondary purpose for this attribute: it can also be used to specify what version of JavaScript is required to interpret a script. When you specify the `language="JavaScript"` attribute for a script, any JavaScript-enabled browser will run the script. Suppose, however, that you have written a script that uses the exception-handling features of JavaScript 1.5. To avoid syntax errors in browsers that do not support this version of the language, you could embed your script with this tag:

```
<script language="JavaScript1.5">
```

If you do this, only browsers that support JavaScript 1.5 (and its exception-handling features) will run the script; any others will ignore it.

The use of the string "JavaScript1.2" in the `language` attribute deserves special mention. When Netscape 4 was being prepared for release, it appeared that the emerging ECMA-262 standard would require some incompatible changes to certain features of the language. To prevent these incompatible changes from breaking existing scripts, the designers of JavaScript at Netscape took the sensible precaution of implementing the changes only when "JavaScript1.2" was explicitly specified in the `language` attribute.

Unfortunately, the ECMA standard was not finalized before Netscape 4 was released, and after the release, the proposed incompatible changes to the language were removed from the standard. Thus, specifying `language="JavaScript1.2"` makes Netscape 4 behave in ways that are not compatible with previous browsers or with the ECMA specification. (See [Section 11.6](#), for complete details on these incompatibilities.) For this reason, you may want to avoid specifying "JavaScript1.2" as a value for the `language` attribute.

12.2.1.2 The `</script>` tag

You may at some point find yourself writing a script that uses the `document.write()` method to output a script into some other browser window or frame. If you do this, you'll need to write out a `</script>` tag to terminate the script you are writing. You must be careful, though -- the HTML parser makes no attempt to understand your JavaScript code, and if it sees the string "`</script>`" in your code, even if it appears within quotes, it assumes that it has found the closing tag of the currently running script. To avoid this problem, simply break up the tag into pieces and write it out using an expression like `"</" + "script>":`

```
<script>
f1.document.write("<script>");
f1.document.write("document.write('<h2>This is the quoted
script</h2>')");
f1.document.write("</" + "script>");
</script>
```

Alternatively, you can escape the `/` in `</script>` with a backslash:

```
f1.document.write("<\/script>");
```

12.2.1.3 The `defer` attribute

The HTML 4 standard defines an attribute of the `<script>` tag that is not yet in common use but is nonetheless important. As I mentioned briefly earlier, a script may call the `document.write()` method to dynamically add content to a document. Because of this, when the HTML parser encounters a script, it must stop parsing the document and wait for the script to execute.

If you write a script that does not produce any document output -- for example, a script that defines a function but never calls `document.write()` -- you may use the `defer` attribute in the `<script>` tag as a hint to the browser that it is safe for it to continue parsing the HTML document and defer execution of the script until it encounters a script that cannot be deferred. Doing this may result in improved performance in browsers that take advantage of the `defer` attribute. Note that `defer` does not have a value; it simply must be present in the tag:

```
<script defer>
```

```
    // Any JavaScript code that does not call document.write( )  
</script>
```

12.2.2 Including JavaScript Files

As of JavaScript 1.1, the `<script>` tag supports a `src` attribute. The value of this attribute specifies the URL of a file containing JavaScript code. It is used like this:

```
<script src="../../javascript/util.js"></script>
```

A JavaScript file typically has a `.js` extension and contains pure JavaScript, without `<script>` tags or any other HTML.

A `<script>` tag with the `src` attribute specified behaves exactly as if the contents of the specified JavaScript file appeared directly between the `<script>` and `</script>` tags. Any code that does appear between these tags is ignored by browsers that support the `src` attribute (although it is still executed by browsers such as Netscape 2 that do not recognize the attribute). Note that the closing `</script>` tag is required even when the `src` attribute is specified and there is no JavaScript between the `<script>` and `</script>` tags.

There are a number of advantages to using the `src` tag:

- It simplifies your HTML files by allowing you to remove large blocks of JavaScript code from them.
- When you have a function or other JavaScript code used by several different HTML files, you can keep it in a single file and read it into each HTML file that needs it. This reduces disk usage and makes code maintenance much easier.
- When JavaScript functions are used by more than one page, placing them in a separate JavaScript file allows them to be cached by the browser, making them load more quickly. When JavaScript code is shared by multiple pages, the time savings of caching more than outweigh the small delay required for the browser to open a separate network connection to download the JavaScript file the first time it is requested.
- Because the `src` attribute takes an arbitrary URL as its value, a JavaScript program or web page from one web server can employ code (such as subroutine libraries) exported by other web servers.

12.2.3 Event Handlers

JavaScript code in a script is executed once, when the HTML file that contains it is read into the web browser. A program that uses only this sort of static script cannot dynamically respond to the user. More dynamic programs define event handlers that are automatically invoked by the web browser when certain events occur -- for example,

when the user clicks on a button within a form. Because events in client-side JavaScript originate from HTML objects (such as buttons), event handlers are defined as attributes of those objects. For example, to define an event handler that is invoked when the user clicks on a checkbox in a form, you specify the handler code as an attribute of the HTML tag that defines the checkbox:

```
<input type="checkbox" name="opts" value="ignore-case"
      onclick="ignore-case = this.checked;"
>
```

What's of interest to us here is the `onclick` attribute.^[3] The string value of the `onclick` attribute may contain one or more JavaScript statements. If there is more than one statement, the statements must be separated from each other with semicolons. When the specified event -- in this case, a click -- occurs on the checkbox, the JavaScript code within the string is executed.

^[3] All HTML event handler attribute names begin with "on".

While you can include any number of JavaScript statements within an event handler definition, a common technique when more than one or two simple statements are required is to define the body of an event handler as a function between `<script>` and `</script>` tags. Then you can simply invoke this function from the event handler. This keeps most of your actual JavaScript code within scripts and reduces the need to mingle JavaScript and HTML.

We'll cover events and event handlers in much more detail in [Chapter 19](#), but you'll see them used in a variety of examples before then. [Chapter 19](#) includes a comprehensive list of event handlers, but these are the most common:

`onclick`

This handler is supported by all button-like form elements, as well as `<a>` and `<area>` tags. It is triggered when the user clicks on the element. If an `onclick` handler returns `false`, the browser does not perform any default action associated with the button or link; for example, it doesn't follow a hyperlink (for an `<a>` tag) or submit a form (for a **Submit** button).

`onmousedown` , `onmouseup`

These two event handlers are a lot like `onclick`, but they are triggered separately when the user presses and releases a mouse button. Document elements that support `onclick` also support these handlers. In IE 4 and Netscape 6, these handlers are actually supported by just about all document elements.

`onmouseover` , `onmouseout`

These two event handlers are triggered when the mouse pointer moves over or out of a document element, respectively. They are used most frequently with `<a>` tags. If the `onmouseover` handler of an `<a>` tag returns `true`, it prevents the browser from displaying the URL of the link in the status line.

`onchange`

This event handler is supported by the `<input>` , `<select>`, and `<textarea>` elements. It is triggered when the user changes the value displayed by the element and then tabs or otherwise moves focus out of the element.

`onsubmit` , `onreset`

These event handlers are supported by the `<form>` tag and are triggered when the form is about to be submitted or reset. They can return `false` to cancel the submission or reset. The `onsubmit` handler is commonly used to perform client-side form validation.

For a realistic example of the use of event handlers, take another look at the interactive loan-payment script in [Example 1-3](#). The HTML form in this example contains a number of event handler attributes. The body of these handlers is simple: they simply call the `calculate()` function defined elsewhere within a `<script>`.

12.2.4 JavaScript in URLs

Another way that JavaScript code can be included on the client side is in a URL following the `javascript:` pseudoprotocol specifier. This special protocol type specifies that the body of the URL is arbitrary JavaScript code to be run by the JavaScript interpreter. If the JavaScript code in a `javascript:` URL contains multiple statements, the statements must be separated from one another by semicolons. Such a URL might look like this:

```
javascript:var now = new Date( ); "<h1>The time is:</h1>" + now;
```

When the browser loads one of these JavaScript URLs, it executes the JavaScript code contained in the URL and uses the string value of the last JavaScript statement as the contents of the new document to display. This string value may contain HTML tags and is formatted and displayed just like any other document loaded into the browser.

JavaScript URLs may also contain JavaScript statements that perform actions but return no value. For example:

```
javascript:alert("Hello World!")
```

When this sort of URL is loaded, the browser executes the JavaScript code, but because there is no value to display as the new document, it does not modify the currently displayed document.

Often, we want to use a `javascript:` URL to execute some JavaScript code without altering the currently displayed document. To do this, you must be sure that the last statement in the URL has no return value. One way to ensure this is to use the `void` operator to explicitly specify an undefined return value. Simply use the statement `void 0;` at the end of your `javascript:` URL. For example, here is a URL that opens a new, blank browser window without altering the contents of the current window:

```
javascript>window.open("about:blank"); void 0;
```

Without the `void` operator in this URL, the return value of the `Window.open()` method call would be converted to a string and displayed, and the current document would be overwritten by a document that appears something like this:

```
[object Window]
```

You can use a `javascript:` URL anywhere you'd use a regular URL. One important way to use this syntax is to type it directly into the **Location** field of your browser, where you can test arbitrary JavaScript code without having to open your editor and create an HTML file containing the code.

`javascript:` URLs can be used in bookmarks, where they form useful mini-JavaScript programs, or "bookmarklets," that can be easily launched from a menu or toolbar of bookmarks.

`javascript:` URLs can also be used as the `href` value of a hyperlink. When the user clicks on such a link, the specified JavaScript code is executed. Or, if you specify a `javascript:` URL as the value of the `action` attribute of a `<form>` tag, the JavaScript code in the URL is executed when the user submits the form. In these contexts, the `javascript:` URL is essentially a substitute for an event handler.

There are a few circumstances where a `javascript:` URL can be used with objects that do not support event handlers. For example, the `<area>` tag does not support an `onclick` event handler on Windows platforms in Netscape 3 (though it does in Netscape 4). So, if

you want to execute JavaScript code when the user clicks on a client-side image map in Netscape 3, you must use a `javascript: URL`.

12.2.5 JavaScript in Nonstandard Contexts

Both Netscape and Microsoft have implemented proprietary extensions in their browsers, and you may occasionally see JavaScript code in a context other than those described here. For example, Internet Explorer allows you to define event handlers in a `<script>` tag that uses special `for` and `event` attributes. Netscape 4 allows you to use JavaScript as an alternative syntax for defining CSS style sheets within a `<style>` tag. Netscape 4 also extends the HTML entity syntax and allows JavaScript to appear within entities (but only within the values of HTML attributes). This can result in HTML that looks like this:

```
<table border="{getBorderWidth( )};">
```

Finally, Netscape 4 also supports a form of conditional comment based on this JavaScript entity syntax. Note that Netscape 6 and the Mozilla browser on which it is based no longer support these nonstandard uses of JavaScript.

12.3 Execution of JavaScript Programs

The previous section discussed the mechanics of integrating JavaScript code into an HTML file. Now we move on to discuss exactly how that integrated JavaScript code is executed by the JavaScript interpreter. The following sections explain how different forms of JavaScript code are executed. While some of this material is fairly obvious, there are a number of important details that are not so obvious.

12.3.1 Scripts

JavaScript statements that appear between `<script>` and `</script>` tags are executed in order of appearance; when more than one script appears in a file, the scripts are executed in the order in which they appear. If a script calls `document.write()`, any text passed to that method is inserted into the document immediately after the closing `</script>` tag and is parsed by the HTML parser when the script finishes running. The same rules apply to scripts included from separate files with the `src` attribute.

The detail that is not so obvious, but is nevertheless important to remember, is that execution of scripts occurs as part of the web browser's HTML parsing process. Thus, if a script appears in the `<head>` section of an HTML document, none of the `<body>` section of the document has been defined yet. This means that the JavaScript objects that represent the contents of the document body, such as Form and Link, have not been created yet and cannot be manipulated by that code.

Your scripts should not attempt to manipulate objects that have not yet been created. For example, you can't write a script that manipulates the contents of an HTML form if the script appears before the form in the HTML file. Some other, similar rules apply on a case-by-case basis. For example, there are properties of the Document object that may be set only from a script in the `<head>` section of an HTML document, before the browser has begun to parse the document content in the `<body>` section. Any special rules of this sort are documented in the reference page for the affected object or property in the client-side reference.

Since scripts are executed while the HTML document that contains them is being parsed and displayed, they should not take too long to run. Because scripts can create dynamic document content with `document.write()`, the HTML parser must stop parsing the document whenever the JavaScript interpreter is running a script. An HTML document cannot be fully displayed until all the scripts it contains have finished executing. If a script performs some computationally intensive task that takes a long time to run, the user may become frustrated waiting for the document to be displayed. Thus, if you need to perform a lot of computation with JavaScript, you should define a function to do the computation and invoke that function from an event handler when the user requests it, rather than doing the computation when the document is first loaded.

As I noted earlier, scripts that use the `src` attribute to read in external JavaScript files are executed just like scripts that include their code directly in the file. What this means is that the HTML parser and the JavaScript interpreter must both stop and wait for the external JavaScript file to be downloaded. (Unlike embedded images, scripts cannot be downloaded in the background while the HTML parser continues to run.) Downloading an external file of JavaScript code, even over a relatively fast modem connection, can cause noticeable delays in the loading and execution of a web page. Of course, once the JavaScript code is cached locally, this problem effectively disappears.

12.3.2 Functions

Remember that defining a function is not the same as executing it. It is perfectly safe to define a function that manipulates objects that have not yet been created. Just take care that the function is not executed or invoked until the necessary variables, objects, and so on all exist. I said earlier that you can't write a script to manipulate an HTML form if the script appears before the form in the HTML file. You can, however, write a script that defines a function to manipulate the form, regardless of the relative locations of the script and form. In fact, this is a common practice. Many JavaScript programs start off with a script in the `<head>` of the document that does nothing more than define functions that are used in the `<body>` of the HTML file.

It is also common to write JavaScript programs that use scripts simply to define functions that are later invoked through event handlers. As we'll see in the next section, you must take care in this case to ensure two things: that all functions are defined before any event handler attempts to invoke them, and that event handlers and the functions they invoke do not attempt to use objects that have not yet been defined.

12.3.3 Event Handlers

Defining an event handler as the value of an `onclick` or another HTML attribute is much like defining a JavaScript function: the code is not immediately executed. Event-handler execution is asynchronous. Since events generally occur when the user interacts with HTML objects, there is no way to predict when an event handler will be invoked.

Event handlers share an important restriction with scripts: they should not take a long time to execute. As we've seen, scripts should run quickly because the HTML parser cannot continue parsing until the script finishes executing. Event handlers, on the other hand, should not take long to run because the user cannot interact with your program until the program has finished handling the event. If an event handler performs some time-consuming operation, it may appear to the user that the program has hung, frozen, or crashed.

If for some reason you must perform a long operation in an event handler, be sure that the user has explicitly requested that operation, and then notify him that there will be a wait. As we'll see in [Chapter 13](#), you can notify the user by posting an `alert()` dialog box or displaying text in the browser's status line. Also, if your program requires a lot of background processing, you can schedule a function to be called repeatedly during idle time with the `setTimeout()` method.

It is important to understand that event handlers may be invoked before a web page is fully loaded and parsed. This is easier to understand if you imagine a slow network connection -- even a half-loaded document may display hypertext links and form elements that the user can interact with, thereby causing event handlers to be invoked before the second half of the document is loaded.

The fact that event handlers can be invoked before a document is fully loaded has two important implications. First, if your event handler invokes a function, you must be sure that the function is already defined before the handler calls it. One way to guarantee this is to define all your functions in the `<head>` section of an HTML document. This section of a document is always completely parsed (and any functions in it defined) before the `<body>` section of the document is parsed. Since all objects that define event handlers must themselves be defined in the `<body>` section, functions in the `<head>` section are guaranteed to be defined before any event handlers are invoked.

The second implication is that you must be sure that your event handler does not attempt to manipulate HTML objects that have not yet been parsed and created. An event handler can always safely manipulate its own object, of course, and also any objects that are defined before it in the HTML file. One strategy is simply to define your web page's user interface in such a way that event handlers refer only to previously defined objects. For example, if you define a form that uses event handlers only on the **Submit** and **Reset** buttons, you just need to place these buttons at the bottom of the form (which is where good user-interface style says they should go anyway).

In more complex programs, you may not be able to ensure that event handlers manipulate only objects defined before them, so you need to take extra care with these programs. If an event handler manipulates only objects defined within the same form, it is pretty unlikely that you'll ever have problems. When you manipulate objects in other forms or other frames, however, this starts to be a real concern. One technique is to test for the existence of the object you want to manipulate before you manipulate it. You can do this simply by comparing it (and any parent objects) to `null`. For example:

```
<script>
function set_name_other_frame(name)
{
    if (parent.frames[1] == null) return;    // Other frame not yet
defined
    if (!parent.frames[1].document) return; // Document not yet loaded
in it
    if (!parent.frames[1].document.myform) return;    // Form not
yet defined
    if (!parent.frames[1].document.myform.name) return; // Field not
yet defined

    parent.frames[1].document.myform.name.value = name;
}
</script>

<input type="text" name="lastname"
        onchange="set_name_other_frame(this.value)";
>
```

In JavaScript 1.5 and later, you can omit the existence tests in the previous code if you instead use the `try/catch` statement to catch the exception that will be thrown if the function is invoked before the document is fully loaded.

Another technique that an event handler can use to ensure that all required objects are defined involves the `onload` event handler. This event handler is defined in the `<body>` or `<frameset>` tag of an HTML file and is invoked when the document or frameset is fully loaded. If you set a flag within the `onload` event handler, other event handlers can test this flag to see if they can safely run, with the knowledge that the document is fully loaded and all objects it contains are defined. For example:

```
<body onload="window.fullyLoaded = true;">
  <form>
    <input type="button" value="Do It!"
           onclick="if (window.fullyLoaded) doit( );">
  </form>
</body>
```

12.3.3.1 *onload and onunload event handlers*

The `onload` event handler and its partner `onunload` are worth a special mention in the context of the execution order of JavaScript programs. Both of these event handlers are defined in the `<body>` or `<frameset>` tag of an HTML file. (No HTML file can legally contain both of these tags.) The `onload` handler is executed when the document or frameset is fully loaded, which means that all images have been downloaded and displayed, all subframes have loaded, any Java applets have started running, and so on. Be aware that when you are working with multiple frames, there is no guarantee of the order in which the `onload` event handler is invoked for the various frames, except that the handler for the parent frame is invoked after the handlers of all its child frames.

The `onunload` handler is executed just before the page is unloaded, which occurs when the browser is about to move on to a new page. You can use it to undo the effects of your `onload` handler or other scripts in your web page. For example, if your web page opens up a secondary browser window, the `onunload` handler provides an opportunity to close that window when the user moves on to some other web page. The `onunload` handler should not run any kind of time-consuming operation, nor should it pop up a dialog box. It exists simply to perform a quick cleanup operation; running it should not slow down or impede the user's transition to a new page.

12.3.4 JavaScript URLs

JavaScript code in a `javascript:` URL is not executed when the document containing the URL is loaded. It is not interpreted until the browser tries to load the document to which the URL refers. This may be when a user types in a JavaScript URL or, more likely, when a user follows a link, clicks on a client-side image map, or submits a form. `javascript:` URLs are often used as an alternative to event handlers, and as with event handlers, the code in those URLs can be executed before a document is fully loaded. Thus, you must take the same precautions with `javascript:` URLs that you take with event handlers to ensure that they do not attempt to reference objects (or functions) that are not yet defined.

12.3.5 Window and Variable Lifetime

A final topic in our investigation of how client-side JavaScript programs run is the issue of variable lifetime. We've seen that the Window object is the global object for client-side JavaScript and that all global variables are properties of the Window object. What happens to Window objects and the variables they contain when the web browser moves from one web page to another?

Whenever a new document is loaded into a window or a frame, the Window object for that window or frame is restored to its default state: any properties and functions defined by a script in the previous document are deleted, and any of the standard system properties that may have been altered or overwritten are restored. Every document begins with a "clean slate." Your scripts can rely on this -- they will not inherit a corrupted environment from the previous document. Any variables and functions your scripts define persist only until the document is replaced with a new one.

The clean slate we're discussing here is the Window object that represents the window or frame into which the document is loaded. As we've discussed, this Window object is the global object for JavaScript code in that window or frame. However, if you're working with multiple frames or multiple windows, a script in one window may refer to the Window objects that represent other windows or frames. So in addition to considering the persistence of variables and functions defined in Window objects, we must also consider the persistence of the Window object itself.

A Window object that represents a top-level browser window exists as long as that window exists. A reference to the Window object remains valid regardless of how many web pages the window loads and unloads. The Window object is valid as long as the top-level window is open.^[4]

^[4] A Window object may not actually be destroyed when its window is closed. If there are still references to the Window object from other windows, the object is not garbage collected. However, a reference to a window that has been closed is of very little practical use.

A Window object that represents a frame remains valid as long as that frame remains within the frame or window that contains it. For example, if frame A contains a script that has a reference to the Window object for frame B, and a new document is loaded into frame B, frame A's reference to the Window object remains valid. Any variables or functions defined in frame B's Window object will be deleted when the new document is loaded, but the Window object itself remains valid (until the containing frame or window loads a new document and overwrites both frame A and frame B).

This means that Window objects, whether they represent top-level windows or frames, are quite persistent. The lifetime of a Window object may be longer than that of the web pages it contains and displays and longer than the lifetime of the scripts contained in the web pages it displays.

Chapter 13. Windows and Frames

[Chapter 12](#) described the Window object and the central role it plays in client-side JavaScript. We've seen that the Window object serves as the global object for client-side JavaScript programs, and, as illustrated in [Figure 12-1](#), it is also the root of the client-side object hierarchy.

Besides these special roles, the Window object is an important object in its own right. Every web browser window and every frame within every window is represented by a Window object. The Window object defines quite a few properties and methods that are important in client-side JavaScript programming. This chapter explores those properties and methods and demonstrates some important techniques for programming with windows and frames. Note that because the Window object is so central to client-side programming, this chapter is quite long. Don't feel you have to master all this material at once -- you may find it easier to study this chapter in several shorter chunks!

13.1 Window Overview

We begin this chapter with an overview of some of the most commonly used properties and methods of the Window object. Later sections of the chapter explain this material in more detail. As usual, the client-side reference section contains complete coverage of Window object properties and methods.

The most important properties of the Window object are the following:

`closed`

A boolean value that is `true` only if the window has been closed.

`defaultStatus, status`

The text that appears in the status line of the browser.

`document`

A reference to the Document object that represents the HTML document displayed in the window. The Document object is covered in detail in [Chapter 14](#).

`frames[]`

An array of Window objects that represent the frames (if any) within the window.

history

A reference to the History object that represents the user's browsing history for the window.

location

A reference to the Location object that represents the URL of the document displayed in the window. Setting this property causes the browser to load a new document.

name

The name of the window. Can be used with the `target` attribute of the HTML `<a>` tag, for example.

opener

A reference to the Window object that opened this one, or `null` if this window was opened by the user.

parent

If the current window is a frame, a reference to the frame of the window that contains it.

self

A self-referential property; a reference to the current Window object. A synonym for `window`.

top

If the current window is a frame, a reference to the Window object of the top-level window that contains the frame. Note that `top` is different from `parent` for frames nested within other frames.

window

A self-referential property; a reference to the current Window object. A synonym for `self`.

The Window object also supports a number of important methods:

`alert()`, `confirm()`, `prompt()`

Display simple dialog boxes to the user and, for `confirm()` and `prompt()`, get the user's response.

`close()`

Close the window.

`focus()`, `blur()`

Request or relinquish keyboard focus for the window. The `focus()` method also ensures that the window is visible by bringing it to the front of the stacking order.

`moveBy()`, `moveTo()`

Move the window.

`open()`

Open a new top-level window to display a specified URL with a specified set of features.

`print()`

Print the window or frame -- same as if the user had selected the **Print** button from the window's toolbar (Netscape 4 and later and IE 5 and later only).

`resizeBy()`, `resizeTo()`

Resize the window.

`scrollBy()`, `scrollTo()`

Scroll the document displayed within the window.

```
setInterval( ), clearInterval( )
```

Schedule or cancel a function to be repeatedly invoked with a specified delay between invocations.

```
setTimeout( ), clearTimeout( )
```

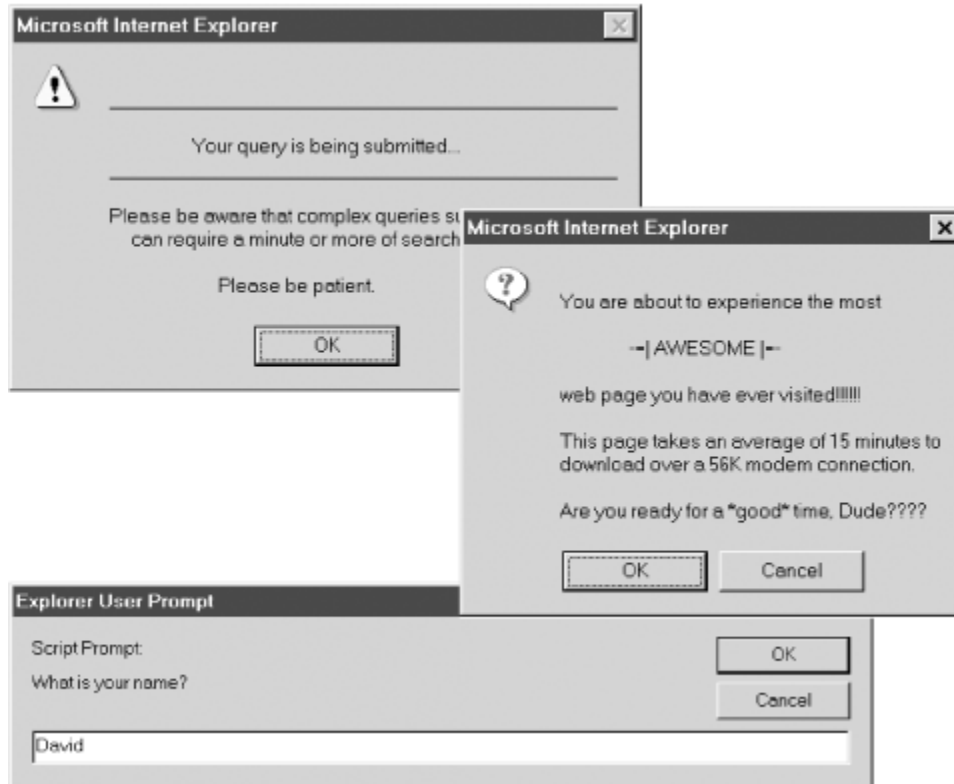
Schedule or cancel a function to be invoked once after a specified number of milliseconds.

As you can see from these lists, the `Window` object provides quite a bit of functionality. The remainder of this chapter explores much of that functionality in more detail.

13.2 Simple Dialog Boxes

Three commonly used `Window` methods are `alert()`, `confirm()`, and `prompt()`. These methods pop up simple dialog boxes. `alert()` displays a message to the user, `confirm()` asks the user to click an **Ok** or **Cancel** button to confirm or cancel an operation, and `prompt()` asks the user to enter a string. Sample dialog boxes produced by these three methods are shown in [Figure 13-1](#).

Figure 13-1. `alert()`, `confirm()`, and `prompt()` dialog boxes



Note that the text displayed by these dialog boxes is plain text, not HTML-formatted text. You can format these dialog boxes only with spaces, newlines, and various punctuation characters. Adjusting the layout generally requires trial and error. Bear in mind, though, that the dialog boxes look different on different platforms and in different browsers, so you can't always count on your formatting to look right on all possible browsers.

Some browsers (such as Netscape 3 and 4) display the word "JavaScript" in the titlebar or upper-left corner of all dialog boxes produced by `alert()`, `confirm()`, and `prompt()`. Although designers find this annoying, it should be considered a feature instead of a bug: it is there to make the origin of the dialog box clear to users and to prevent you from writing Trojan-horse code that spoofs system dialog boxes and tricks users into entering their passwords or doing other things that they shouldn't do.

The `confirm()` and `prompt()` methods *block* -- that is, those methods do not return until the user dismisses the dialog boxes they display. This means that when you pop up one of these boxes, your code stops running and the currently loading document, if any, stops loading until the user responds with the requested input. There is no alternative to blocking for these methods -- their return value is the user's input, so they must wait for the user before they can return. In most browsers, the `alert()` method also blocks and waits for the user to dismiss the dialog box. In some browsers, however (notably Netscape 3 and 4 on Unix platforms), `alert()` does not block. In practice, this minor incompatibility rarely causes problems.

[Example 13-1](#) shows some typical uses of these methods.

Example 13-1. Using the `alert()`, `confirm()`, and `prompt()` methods

```
// Here's a function that uses the alert( ) method to tell the user
// that form submission will take some time and that the user should
// be patient. It would be suitable for use in the onsubmit event
handler
// of an HTML form.
// Note that all formatting is done with spaces, newlines, and
underscores.
function warn_on_submit( )
{
    alert("\n_____ _\n\n"
+
        "                Your query is being submitted...\n" +
        "_____ _\n\n"
+
        "Please be aware that complex queries such as yours\n" +
        "    can require a minute or more of search time.\n\n" +
        "                Please be patient.");
}

// Here is a use of the confirm( ) method to ask if the user really
// wants to visit a web page that takes a long time to download. Note
that
// the return value of the method indicates the user response. Based
// on this response, we reroute the browser to an appropriate page.
var msg = "\nYou are about to experience the most\n\n" +
    "                -=| AWESOME |=-\n\n" +
    "web page you have ever visited!!!!!!\n\n" +
    "This page takes an average of 15 minutes to\n" +
    "download over a 56K modem connection.\n\n" +
    "Are you ready for a *good* time, Dude?????";

if (confirm(msg))
    location.replace("awesome_page.html");
else
    location.replace("lame_page.html");

// Here's some very simple code that uses the prompt( ) method to get
// a user's name and then uses that name in dynamically generated HTML.
n = prompt("What is your name?", "");
document.write("<hr><h1>Welcome to my home page, " + n + "</h1><hr>");
```

13.3 The Status Line

Web browsers typically display a *status line* at the bottom of every window (except for those explicitly created without one), where the browser can display messages to the user. When the user moves the mouse over a hypertext link, for example, the browser usually displays the URL to which the link points. And when the user moves the mouse over a browser control button, the browser may display a simple context help message that

explains the purpose of the button. You can also make use of this status line in your own programs. Its contents are controlled by two properties of the Window object: `status` and `defaultStatus`.

Although web browsers usually display the URL of a hypertext link when the user passes the mouse pointer over the link, you may have encountered some links that don't behave this way -- links that display some text other than the link's URL. This effect is achieved with the `status` property of the Window object and the `onmouseover` event handler of hypertext links:

```
<!-- Here's how you set the status line in a hyperlink.
-- Note that the event handler *must* return true for this to work. -
->
Lost? Dazed and confused? Visit the
<a href="sitemap.html" onmouseover="status='Go to Site Map'; return
true;">
  Site Map
</a>

<!-- You can do the same thing for client-side image maps -->

<map name="map1">
  <area coords="0,0,50,20" href="info.html"
    onmouseover="status='Visit our Information Center'; return true;">
  <area coords="0,20,50,40" href="order.html"
    onmouseover="status='Place an order'; return true;">
  <area coords="0,40,50,60" href="help.html"
    onmouseover="status='Get help fast!'; return true;">
</map>
```

The `onmouseover` event handler in this example must return `true`. This tells the browser that it should not perform its own default action for the event -- that is, it should not display the URL of the link in the status line. If you forget to return `true`, the browser overwrites whatever message the handler displays in the status line with its own URL. Don't worry if you do not fully understand the event handler in this example. We'll explain events in [Chapter 19](#).

When the user moves the mouse pointer over a hyperlink, the browser displays the URL for the link, then erases the URL when the mouse moves off the hyperlink. The same is true when you use an `onmouseover` event handler to set the Window `status` property -- your custom message is displayed while the mouse is over the hyperlink and is erased when the mouse moves off the link.

The `status` property is intended for exactly the sort of transient message we saw in the previous example. Sometimes, though, you want to display a message that is not so transient in the status line -- for example, you might display a welcome message to users visiting your web page or a simple line of help text for novice visitors. To do this, you set the `defaultStatus` property of the Window object; this property specifies the default text displayed in the status line. That text is temporarily replaced with URLs, context help

messages, or other transient text when the mouse pointer is over hyperlinks or browser control buttons, but once the mouse moves off those areas, the default text is restored.

You might use the `defaultStatus` property like this to provide a friendly and helpful message to real beginners:

```
<script>
defaultStatus = "Welcome!  Click on underlined blue text to navigate.";
</script>
```

13.4 Timeouts and Intervals

The `setTimeout()` method of the `Window` object schedules a piece of JavaScript code to be run at some specified time in the future. The `clearTimeout()` method can be used to cancel the execution of that code. `setTimeout()` is commonly used to perform animations or other kinds of repetitive actions. If a function runs and then uses `setTimeout()` to schedule itself to be called again, we get a process that repeats without any user intervention. JavaScript 1.2 has added the `setInterval()` and `clearInterval()` methods, which are like `setTimeout()` and `clearTimeout()`, except that they automatically reschedule the code to run repeatedly; there is no need for the code to reschedule itself.

The `setTimeout()` method is commonly used in conjunction with the `status` or `defaultStatus` properties to animate some kind of message in the status bar of the browser. In general, animations involving the status bar are gaudy, and you should shun them! There are, however, a few status-bar animation techniques that can be useful and in good taste. [Example 13-2](#) shows such a tasteful status-bar animation. It displays the current time in the status bar and updates that time once a minute. Because the update occurs only once a minute, this animation does not produce a constant flickering distraction at the bottom of the browser window, like so many others do.

Note the use of the `onload` event handler of the `<body>` tag to perform the first call to the `display_time_in_status_line()` method. This event handler is invoked once when the HTML document is fully loaded into the browser. After this first call, the method uses `setTimeout()` to schedule itself to be called every 60 seconds so that it can update the displayed time.

Example 13-2. A digital clock in the status line

```
<html>
<head>
<script>
// This function displays the time in the status line
// Invoke it once to activate the clock; it will call itself from then
on
function display_time_in_status_line( )
{
```

```

    var d = new Date( ); // Get the current time
    var h = d.getHours( ); // Extract hours: 0 to 23
    var m = d.getMinutes( ); // Extract minutes: 0 to 59
    var ampm = (h >= 12)?"PM":"AM"; // Is it a.m. or p.m.?
    if (h > 12) h -= 12; // Convert 24-hour format to
12-hour
    if (h == 0) h = 12; // Convert 0 o'clock to
midnight
    if (m < 10) m = "0" + m; // Convert 0 minutes to 00
minutes, etc.
    var t = h + ':' + m + ' ' + ampm; // Put it all together

    defaultStatus = t; // Display it in the status
line

    // Arrange to do it all again in one minute
    setTimeout("display_time_in_status_line( )", 60000); // 60000 ms
is one minute
}
</script>
</head>
<!-- Don't bother starting the clock till everything is loaded. The
-- status line will be busy with other messages during loading,
anyway. -->
<body onload="display_time_in_status_line( );">
<!-- The HTML document contents go here -->
</body>
</html>

```

In JavaScript 1.2, [Example 13-2](#) could be written using `setInterval()` instead of `setTimeout()`. In this case, the `setTimeout()` call would be removed from the `display_time_in_status_line()` method, and we'd remove the `onload` event handler. Instead, after defining `display_time_in_status_line()`, our script would call `setInterval()` to schedule an invocation of the function that automatically repeats once every 60,000 milliseconds.

13.5 Error Handling

The `onerror` property of a Window object is special. If you assign a function to this property, the function will be invoked whenever a JavaScript error occurs in that window: the function you assign becomes an error handler for the window.

Three arguments are passed to an error handler. The first is a message describing the error that occurred. This may be something like "missing operator in expression", "self is read-only", or "myname is not defined". The second argument is a string that contains the URL of the document containing the JavaScript code that caused the error. The third argument is the line number within the document where the error occurred. An error handler can use these arguments for any purpose it desires. A typical error handler might display the error message to the user, log it somewhere, or force the error to be ignored.

In addition to those three arguments, the return value of the `onerror` handler is significant. Browsers typically display an error message in a dialog box or in the status line when an error occurs. If the `onerror` handler returns `true`, it tells the system that the handler has handled the error and that no further action is necessary -- in other words, the system should not display its own error message. For example, if you do not want your users to be pestered by error messages, no matter how buggy the code you write is, you could use a line of code like this at the start of all your JavaScript programs:

```
self.onerror = function( ) { return true; }
```

Of course, doing this will make it very difficult for users to give you feedback when your programs fail silently without producing error messages.

We'll see a sample use of an `onerror` handler in [Example 14-1](#). That example uses the `onerror` handler to display the error details to the user and allow the user to submit a bug report containing those details.

Note that the `onerror` error handler is buggy in Netscape 6. Although the function you specify is triggered when an error occurs, the three arguments that are passed are incorrect and unusable. Netscape 6 and other browsers that support JavaScript 1.5 have an alternative means of catching and handling errors, however: they can use the `try/catch` statement. (See [Chapter 6](#) for details.)

13.6 The Navigator Object

The `Window.navigator` property refers to a Navigator object that contains information about the web browser as a whole, such as the version and a list of the data formats it can display. The Navigator object is named after Netscape Navigator, but it is also supported by Internet Explorer. IE also supports `clientInformation` as a vendor-neutral synonym for `navigator`. Unfortunately, Netscape and Mozilla do not support this property.

The Navigator object has five main properties that provide version information about the browser that is running:

`appName`

The simple name of the web browser.

`appVersion`

The version number and/or other version information for the browser. Note that this should be considered an "internal" version number, since it does not always correspond to the version number displayed to the user. For example, Netscape 6

reports a version number of 5.0, since there never was a Netscape 5 release. Also, IE Versions 4 through 6 all report a version number of 4.0, to indicate compatibility with the baseline functionality of fourth-generation browsers.

`userAgent`

The string that the browser sends in its `USER-AGENT` HTTP header. This property typically contains all the information in both `appName` and `appVersion`.

`appName`

The code name of the browser. Netscape uses the code name "Mozilla" as the value of this property. For compatibility, IE does the same thing.

`platform`

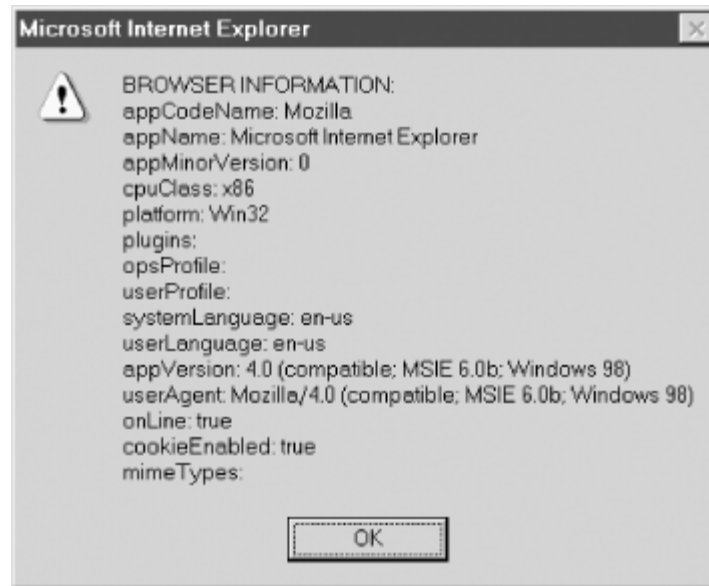
The hardware platform on which the browser is running. This property was added in JavaScript 1.2.

The following lines of JavaScript code display each of these Navigator object properties in a dialog box:

```
var browser = "BROWSER INFORMATION:\n";
for(var propName in navigator) {
    browser += propName + ": " + navigator[propName] + "\n"
}
alert(browser);
```

[Figure 13-2](#) shows the dialog box displayed when the code is run on IE 6.

Figure 13-2. Navigator object properties



As you can see from [Figure 13-2](#), the properties of the Navigator object have values that are sometimes more complex than we are interested in. We are often interested in only the first digit of the `appVersion` property, for example. When using the Navigator object to test browser information, we often use methods such as `parseInt()` and `String.indexOf()` to extract only the information we want. [Example 13-3](#) shows some code that does this: it processes the properties of the Navigator object and stores them in an object named `browser`. These properties, in their processed form, are easier to use than the raw `navigator` properties. The general term for code like this is a "client sniffer," and you can find more complex and general-purpose sniffer code on the Internet.^[1] For many purposes, however, something as simple as that shown in [Example 13-3](#) works just fine.

^[1] See, for example, http://www.mozilla.org/docs/web-developer/sniffer/browser_type.html.

Example 13-3. Determining browser vendor and version

```
/*
 * File: browser.js
 * Include with: <script SRC="browser.js"></script>
 *
 * A simple "sniffer" that determines browser version and vendor.
 * It creates an object named "browser" that is easier to use than
 * the "navigator" object.
 */
// Create the browser object
var browser = new Object( );

// Figure out the browser's major version
browser.version = parseInt(navigator.appVersion);
// Now figure out if the browser is from one of the two
// major browser vendors. Start by assuming it is not.
browser.isNetscape = false;
browser.isMicrosoft = false;
```

```
if (navigator.appName.indexOf("Netscape") != -1)
    browser.isNetscape = true;
else if (navigator.appName.indexOf("Microsoft") != -1)
    browser.isMicrosoft = true;
```

13.7 The Screen Object

In JavaScript 1.2, the `screen` property of a `Window` object refers to a `Screen` object that provides information about the size of the user's display and the number of colors available on it. The `width` and `height` properties specify the size of the display in pixels. The `availWidth` and `availHeight` properties specify the display size that is actually available; they exclude the space required by features such as the Windows taskbar. You can use these properties to help you decide what size images to include in a document, for example, or what size windows to create in a program that creates multiple browser windows.

The `colorDepth` property specifies the base-2 logarithm of the number of colors that can be displayed. Often, this value is the same as the number of bits per pixel used by the display. For example, an 8-bit display can display 256 colors, and if all of these colors were available for use by the browser, the `screen.colorDepth` property would be 8. In some circumstances, however, the browser may restrict itself to a subset of the available colors, and you might find a `screen.colorDepth` value that is lower than the bits-per-pixel value of the screen. If you have several versions of an image that were defined using different numbers of colors, you can test this `colorDepth` property to decide which version to include in a document.

[Example 13-4](#), later in this chapter, shows how the `Screen` object can be used.

13.8 Window Control Methods

The `Window` object defines several methods that allow high-level control of the window itself. The following sections explore how these methods allow us to open and close windows, control window position and size, request and relinquish keyboard focus, and scroll the contents of a window. We conclude with an example that demonstrates several of these features.

13.8.1 Opening Windows

You can open a new web browser window with the `open()` method of the `Window` object. This method takes four optional arguments and returns a `Window` object that represents the newly opened window. The first argument to `open()` is the URL of the document to display in the new window. If this argument is omitted (or is `null` or the empty string), the window will be empty.

The second argument to `open()` is the name of the window. As we'll discuss later in the chapter, this name can be useful as the value of the `target` attribute of a `<form>` or `<a>` tag. If you specify the name of a window that already exists, `open()` simply returns a reference to that existing window, rather than opening a new one.

The third optional argument to `open()` is a list of features that specify the window size and GUI decorations. If you omit this argument, the new window is given a default size and has a full set of standard features: a menu bar, status line, toolbar, and so on. On the other hand, if you specify this argument, you can explicitly specify the size of the window and the set of features it includes. For example, to open a small, resizable browser window with a status bar but no menu bar, toolbar, or location bar, you could use the following line of JavaScript:

```
var w = window.open("smallwin.html", "smallwin",
                    "width=400,height=350,status=yes,resizable=yes");
```

Note that when you specify this third argument, any features you do not explicitly specify are omitted. See `Window.open()` in the client-side reference section for the full set of available features and their names.

The fourth argument to `open()` is useful only when the second argument names an already existing window. This fourth argument is a boolean value that specifies whether the URL specified as the first argument should replace the current entry in the window's browsing history (`true`) or create a new entry in the window's browsing history (`false`), which is the default behavior.

The return value of the `open()` method is the `Window` object that represents the newly created window. You can use this `Window` object in your JavaScript code to refer to the new window, just as you use the implicit `Window` object `window` to refer to the window within which your code is running. But what about the reverse situation? What if JavaScript code in the new window wants to refer back to the window that opened it? In JavaScript 1.1 and later, the `opener` property of a window refers to the window from which it was opened. If the window was created by the user instead of by JavaScript code, the `opener` property is `null`.

An important point about the `open()` method is that it is almost always invoked as `window.open()`, even though `window` refers to the global object and should therefore be entirely optional. `window` is explicitly specified because the `Document` object also has an `open()` method, so specifying `window.open()` helps to make it very clear what we are trying to do. This is not just a helpful habit; it is required in some circumstances, because, as we'll learn in [Chapter 19](#), event handlers execute in the scope of the object that defines them. When the event handler of an HTML button executes, for example, the scope chain includes the `Button` object, the `Form` object that contains the button, the `Document` object that contains the form, and, finally, the `Window` object that contains the document. Thus, if such an event handler refers merely to the `open()` method, this identifier ends up

being resolved in the Document object, and the event handler opens a new document rather than opening a new window!

We'll see the `open()` method in use in [Example 13-4](#).

13.8.2 Closing Windows

Just as the `open()` method opens a new window, the `close()` method closes one. If we've created a Window object `w`, we can close it with:

```
w.close( );
```

JavaScript code running within that window itself could close it with:

```
window.close( );
```

Again, note the explicit use of the `window` identifier to disambiguate the `close()` method of the Window object from the `close()` method of the Document object.

Most browsers allow you to automatically close only those windows that your own JavaScript code has created. If you attempt to close any other window, the user is presented with a dialog box that asks him to confirm (or cancel) that request to close the window. This precaution prevents inconsiderate scripters from writing code to close a user's main browsing window.

In JavaScript 1.1 and later, a Window object continues to exist after the window it represents has been closed. You should not attempt to use any of its properties or methods, however, except to test the `closed` property. This property is `true` if the window has been closed. Remember that the user can close any window at any time, so to avoid errors, it is a good idea to check periodically that the window you are trying to use is still open. We'll see this done in [Example 13-4](#).

13.8.3 Window Geometry

In JavaScript 1.2, `moveTo()` moves the upper-left corner of the window to the specified coordinates. Similarly, `moveBy()` moves the window a specified number of pixels left or right and up or down. `resizeTo()` and `resizeBy()` resize the window by an absolute or relative amount; they are also new in JavaScript 1.2. Note that in order to prevent security attacks that rely on code running in small or offscreen windows that the user does not notice, browsers may restrict your ability to move windows offscreen or to make them too small.

13.8.4 Keyboard Focus and Visibility

The `focus()` and `blur()` methods also provide high-level control over a window. Calling `focus()` requests that the system give keyboard focus to the window, and `blur()` relinquishes keyboard focus. In addition, the `focus()` method ensures that the window is visible by moving it to the top of the stacking order. When you use the `Window.open()` method to open a new window, the browser automatically creates that window on top. But if the second argument specifies the name of a window that already exists, the `open()` method does not automatically make that window visible. Thus, it is common practice to follow calls to `open()` with a call to `focus()`.

`focus()` and `blur()` are defined in JavaScript 1.1 and later.

13.8.5 Scrolling

The Window object also contains methods that scroll the document within the window or frame. `scrollBy()` scrolls the document displayed in the window by a specified number of pixels left or right and up or down. `scrollTo()` scrolls the document to an absolute position. It moves the document so that the specified document coordinates are displayed in the upper-left corner of the document area within the window. These two methods are defined in JavaScript 1.2. In JavaScript 1.1, the `scroll()` method performs the same function as the JavaScript 1.2 `scrollTo()` method. `scrollTo()` is the preferred method, but the `scroll()` method remains for backward compatibility.

In JavaScript 1.2, the elements of the `anchors[]` array of the Document object are Anchor objects. Each Anchor object has `x` and `y` properties that specify the location of the anchor within the document. Thus, you can use these values in conjunction with the `scrollTo()` method to scroll to known locations within the document. Alternatively, in IE 4 and later and Netscape 6 and later, document elements all define a `focus()` method. Invoking this method on an element causes the document to scroll as needed to ensure that the element is visible.

13.8.6 Window Methods Example

[Example 13-4](#) demonstrates the Window `open()`, `close()`, and `moveTo()` methods and several other window-programming techniques that we've discussed. It creates a new window and then uses `setInterval()` to repeatedly call a function that moves it around the screen. It determines the size of the screen with the Screen object and then uses this information to make the window bounce when it reaches any edge of the screen.

Example 13-4. Moving a window

```
<script>
// Here are the initial values for our animation
var x = 0, y = 0, w=200, h=200; // Window position and size
var dx = 5, dy = 5;           // Window velocity
var interval = 100;           // Milliseconds between updates
```

```

// Create the window that we're going to move around
// The javascript: URL is simply a way to display a short document
// The final argument specifies the window size
var win = window.open('javascript:"<h1>BOUNCE!</h1>"', "",
                      "width=" + w + ",height=" + h);

// Set the initial position of the window
win.moveTo(x,y);

// Use setInterval( ) to call the bounce( ) method every interval
// milliseconds. Store the return value so that we can stop the
// animation by passing it to clearInterval( ).
var intervalID = window.setInterval("bounce( )", interval);

// This function moves the window by (dx, dy) every interval ms
// It bounces whenever the window reaches the edge of the screen
function bounce( ) {
    // If the user closed the window, stop the animation
    if (win.closed) {
        clearInterval(intervalID);
        return;
    }

    // Bounce if we have reached the right or left edge
    if ((x+dx > (screen.availWidth - w)) || (x+dx < 0)) dx = -dx;

    // Bounce if we have reached the bottom or top edge
    if ((y+dy > (screen.availHeight - h)) || (y+dy < 0)) dy = -dy;

    // Update the current position of the window
    x += dx;
    y += dy;

    // Finally, move the window to the new position
    win.moveTo(x,y);
}
</script>

<!-- Clicking this button stops the animation! -->
<form>
<input type="button" value="Stop"
        onclick="clearInterval(intervalID); win.close( );">
</form>

```

13.9 The Location Object

The `location` property of a window is a reference to a Location object -- a representation of the URL of the document currently being displayed in that window. The `href` property of the Location object is a string that contains the complete text of the URL. Other properties of this object, such as `protocol`, `host`, `pathname`, and `search`, specify the various individual parts of the URL.

The `search` property of the `Location` object is an interesting one. It contains any portion of a URL following (and including) a question mark. This is often some sort of query string. In general, the question-mark syntax in a URL is a technique for embedding arguments in the URL. While these arguments are usually intended for CGI scripts run on a server, there is no reason why they cannot also be used in JavaScript-enabled pages.

[Example 13-5](#) shows the definition of a general-purpose `getArgs()` function that you can use to extract arguments from the `search` property of a URL. It also shows how this `getArgs()` method could have been used to set initial values of the bouncing window animation parameters in [Example 13-4](#).

Example 13-5. Extracting arguments from a URL

```
/*
 * This function parses comma-separated name=value argument pairs from
 * the query string of the URL. It stores the name=value pairs in
 * properties of an object and returns that object.
 */
function getArgs( ) {
    var args = new Object( );
    var query = location.search.substring(1); // Get query string
    var pairs = query.split(","); // Break at comma
    for(var i = 0; i < pairs.length; i++) {
        var pos = pairs[i].indexOf('='); // Look for
"name=value"
        if (pos == -1) continue; // If not found, skip
        var argname = pairs[i].substring(0,pos); // Extract the name
        var value = pairs[i].substring(pos+1); // Extract the value
        args[argname] = unescape(value); // Store as a
property
        // In JavaScript 1.5, use decodeURIComponent( ) instead of
unescape( )
    }
    return args; // Return the object
}

/*
 * We could have used getArgs( ) in the previous bouncing window
example
 * to parse optional animation parameters from the URL
 */
var args = getArgs( ); // Get arguments
if (args.x) x = parseInt(args.x); // If arguments are
defined...
if (args.y) y = parseInt(args.y); // override default values
if (args.w) w = parseInt(args.w);
if (args.h) h = parseInt(args.h);
if (args.dx) dx = parseInt(args.dx);
if (args.dy) dy = parseInt(args.dy);
if (args.interval) interval = parseInt(args.interval);
```

In addition to its properties, the `Location` object can be used as if it were itself a primitive string value. If you read the value of a `Location` object, you get the same string as you would if you read the `href` property of the object (because the `Location` object has a

suitable `toString()` method). What is far more interesting, though, is that you can assign a new URL string to the `location` property of a window. Assigning a URL to the `Location` object this way has an important side effect: it causes the browser to load and display the contents of the URL you assign. For example, you might assign a URL to the `location` property like this:

```
// If the user is using an old browser that can't display DHTML
content,
// redirect to a page that contains only static HTML
if (parseInt(navigator.appVersion) < 4)
    location = "staticpage.html";
```

As you can imagine, making the browser load specified web pages into windows is a very important programming technique. While you might expect there to be a method you can call to make the browser display a new web page, assigning a URL to the `location` property of a window is the supported technique for accomplishing this end. [Example 13-6](#), later in this chapter, includes an example of setting the `location` property.

Although the `Location` object does not have a method that serves the same function as assigning a URL directly to the `location` property of a window, this object does support two methods (added in JavaScript 1.1). The `reload()` method reloads the currently displayed page from the web server. The `replace()` method loads and displays a URL that you specify. But invoking this method for a given URL is different than assigning that URL to the `location` property of a window. When you call `replace()`, the specified URL replaces the current one in the browser's history list, rather than creating a new entry in that history list. Therefore, if you use `replace()` to overwrite one document with a new one, the **Back** button does not take the user back to the original document, as it does if you load the new document by assigning a URL to the `location` property. For web sites that use frames and display a lot of temporary pages (perhaps generated by a CGI script), using `replace()` is often useful. Since temporary pages are not stored in the history list, the **Back** button is more useful to the user.

Finally, don't confuse the `location` property of the `Window` object, which refers to a `Location` object, with the `location` property of the `Document` object, which is simply a read-only string with none of the special features of the `Location` object.

`document.location` is a synonym for `document.URL`, which in JavaScript 1.1 is the preferred name for this property (because it avoids the potential confusion). In most cases, `document.location` is the same as `location.href`. When there is a server redirect, however, `document.location` contains the URL as loaded, and `location.href` contains the URL as originally requested.

13.10 The History Object

The `history` property of the `Window` object refers to a `History` object for the window. The `History` object was originally designed to model the browsing history of a window as

an array of recently visited URLs. This turned out to be a poor design choice, however; for important security and privacy reasons, it is almost never appropriate to give a script access to the list of web sites that the user has previously visited. Thus, the array elements of the History object are never actually accessible to scripts (except when the user has granted permission to a signed script in Netscape 4 and later). The `length` property of the History object is accessible, but it does not provide any useful information.

Although its array elements are inaccessible, the History object supports three methods (which can be used by normal, unsigned scripts in all browser versions). The `back()` and `forward()` methods move backward or forward in a window's (or frame's) browsing history, replacing the currently displayed document with a previously viewed one. This is similar to what happens when the user clicks on the **Back** and **Forward** browser buttons. The third method, `go()`, takes an integer argument and can skip forward or backward in the history list by multiple pages. Unfortunately, `go()` suffers from bugs in Netscape 2 and 3 and has incompatible behavior in Internet Explorer 3; it is best avoided prior to fourth-generation browsers.

[Example 13-6](#) shows how you might use the `back()` and `forward()` methods of the History and Location objects to add a navigation bar to a framed web site. [Figure 13-3](#) shows what a navigation bar looks like. Note that the example uses JavaScript with multiple frames, which is something we will discuss shortly. It also contains a simple HTML form and uses JavaScript to read and write values from the form. This behavior is covered in detail in [Chapter 15](#).

Figure 13-3. A navigation bar



Example 13-6. A navigation bar using the History and Location objects

<!-- This file implements a navigation bar, designed to go in a frame at

the bottom of a window. Include it in a frameset like the following:

```
<frameset rows="*,75">
  <frame src="about:blank">
  <frame src="navigation.html">
</frameset>
```

-->

```
<script>
```

```
// The function is invoked by the Back button in our navigation bar
```

```
function go_back( )
```

```
{
```

```
  // First, clear the URL entry field in our form
```

```
  document.navbar.url.value = "";
```

```
  // Then use the History object of the main frame to go back
```

```
  parent.frames[0].history.back( );
```

```
  // Wait a second, and then update the URL entry field in the form
  // from the location.href property of the main frame. The wait
  seems
```

```
  // to be necessary to allow the location.href property to get in
  sync.
```

```
  setTimeout("document.navbar.url.value =
parent.frames[0].location.href;",
            1000);
```

```
}
```

```
// This function is invoked by the Forward button in the navigation
bar;
```

```
// it works just like the previous one
```

```
function go_forward( )
```

```
{
```

```
  document.navbar.url.value = "";
```

```
  parent.frames[0].history.forward( );
```

```
  setTimeout("document.navbar.url.value =
parent.frames[0].location.href;",
            1000);
```

```
}
```

```
// This function is invoked by the Go button in the navigation bar and
also
```

```
// when the form is submitted (when the user hits the Return key)
```

```
function go_to( )
```

```
{
```

```
  // Just set the location property of the main frame to the URL
  // the user typed in
```

```
  parent.frames[0].location = document.navbar.url.value;
```

```
}
```

```
</script>
```

<!-- Here's the form, with event handlers that invoke the functions
above -->

```
<form name="navbar" onsubmit="go_to( ); return false;">
  <input type="button" value="Back" onclick="go_back( );">
  <input type="button" value="Forward" onclick="go_forward( );">
  URL:
  <input type="text" name="url" size="50">
  <input type="button" value="Go" onclick="go_to( );">
</form>
```

13.11 Multiple Windows and Frames

Most of the client-side JavaScript examples we've seen so far have involved only a single window or frame. In the real world, JavaScript applications often involve multiple windows or frames. Recall that frames within a window are represented by Window objects; JavaScript makes little distinction between windows and frames. In the most interesting applications, there is JavaScript code that runs independently in each of several windows. The next section explains how the JavaScript code in each window can interact and cooperate with each of the other windows and with the scripts running in each of those windows.

13.11.1 Relationships Between Frames

We've already seen that the `open()` method of the Window object returns a new Window object representing the newly created window. We've also seen that this new window has an `opener` property that refers back to the original window. In this way, the two windows can refer to each other, and each can read properties and invoke methods of the other. The same thing is possible with frames. Any frame in a window can refer to any other frame through the use of the `frames`, `parent`, and `top` properties of the Window object.

Every window has a `frames` property. This property refers to an array of Window objects, each of which represents a frame contained within the window. (If a window does not have any frames, the `frames[]` array is empty and `frames.length` is zero.) Thus, a window (or frame) can refer to its first subframe as `frames[0]`, its second subframe as `frames[1]`, and so on. Similarly, JavaScript code running in a window can refer to the third subframe of its second frame like this:

```
frames[1].frames[2]
```

Every window also has a `parent` property, which refers to the Window object in which it is contained. Thus, the first frame within a window might refer to its sibling frame (the second frame within the window) like this:

```
parent.frames[1]
```

If a window is a top-level window and not a frame, `parent` simply refers to the window itself:

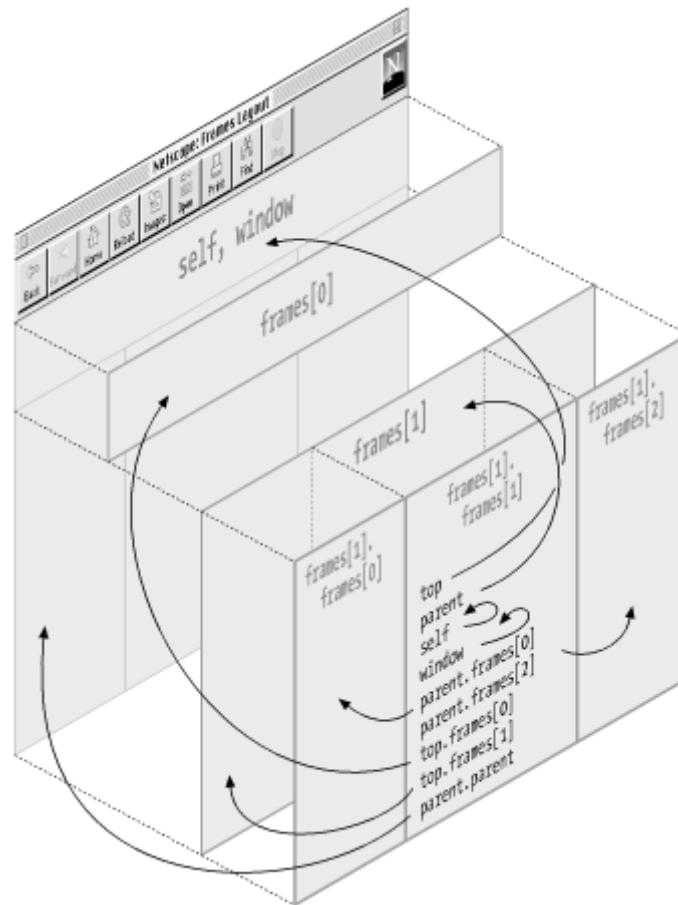
```
parent == self; // For any top-level window
```

If a frame is contained within another frame that is contained within a top-level window, that frame can refer to the top-level window as `parent.parent`. The `top` property is a general-case shortcut, however: no matter how deeply a frame is nested, its `top` property refers to the top-level containing window. If a Window object represents a top-level window, `top` simply refers to that window itself. For frames that are direct children of a top-level window, the `top` property is the same as the `parent` property.

Frames are typically created with `<frameset>` and `<frame>` tags. In HTML 4, however, as implemented in IE 4 and later and Netscape 6 and later, the `<iframe>` tag can also be used to create an "inline frame" within a document. As far as JavaScript is concerned, frames created with `<iframe>` are the same as frames created with `<frameset>` and `<frame>`. Everything discussed here applies to both kinds of frames.

[Figure 13-4](#) illustrates these relationships between frames and shows how code running in any one frame can refer to any other frame through the use of the `frames`, `parent`, and `top` properties. The figure shows a browser window that contains two frames, one on top of the other. The second frame (the larger one on the bottom) itself contains three subframes, side by side.

Figure 13-4. Relationships between frames



With this understanding of the relationships between frames, you may want to revisit [Example 13-6](#), paying particular attention this time to the way the code (which is written to run in a second frame) refers to the `history` and `location` properties of the first frame.

13.11.2 Window and Frame Names

The second, optional argument to the `open()` method discussed earlier is a name for the newly created window. When you create a frame with the HTML `<frame>` tag, you can specify a name with the `name` attribute. An important reason to specify names for windows and frames is that those names can be used as the value of the `target` attribute of the `<a>`, `<map>`, and `<form>` tags. This value tells the browser where you want to display the results of activating a link, clicking on an image map, or submitting a form.

For example, if you have two windows, one named `table_of_contents` and the other `mainwin`, you might have HTML like the following in the `table_of_contents` window:

```
<a href="chapter01.html" target="mainwin">
  Chapter 1, Introduction
</a>
```

The browser loads the specified URL when the user clicks on this hyperlink, but instead of displaying the URL in the same window as the link, it displays it in the window named `mainwin`. If there is no window with the name `mainwin`, clicking the link creates a new window with that name and loads the specified URL into it.

The `target` and `name` attributes are part of HTML and operate without the intervention of JavaScript, but there are also JavaScript-related reasons to give names to your frames. We've seen that every Window object has a `frames[]` array that contains references to each of its frames. This array contains all the frames in a window (or frame), whether or not they have names. If a frame is given a name, however, a reference to that frame is also stored in a new property of the parent Window object. The name of that new property is the same as the name of the frame. Therefore, you might create a frame with HTML like this:

```
<frame name="table_of_contents" src="toc.html">
```

Now you can refer to that frame from another, sibling frame with:

```
parent.table_of_contents
```

This makes your code easier to read and understand than using (and relying on) a hardcoded array index, as you'd have to do with an unnamed frame:

```
parent.frames[1]
```

The `name` property of any Window object contains the name of that window. In JavaScript 1.0, this property is read-only. In JavaScript 1.1 and later, however, you can set this property, thereby changing the name of a window or a frame. One common reason to do this is to set the name of the initial browser window. When a browser starts up, the initial window has no name, so it cannot be used with the `target` attribute. If you set the `name` property of the window, however, you can then use that name in `target` attributes.

13.11.3 JavaScript in Interacting Windows

Recall what we learned in [Chapter 12](#): the Window object serves as the global object for client-side JavaScript code, and the window serves as the execution context for all JavaScript code it contains. This holds true for frames as well: every frame is an independent JavaScript execution context. Because every Window object is its own global object, each window defines its own namespace and its own set of global variables. When viewed from the perspective of multiple frames or windows, global variables do not seem all that global, after all!

Although each window and frame defines an independent JavaScript execution context, this does not mean that JavaScript code running in one window is isolated from code running in other windows. Code running in one frame has a different Window object at the top of its scope chain than code running in another frame. However, the code from both frames is executed by the same JavaScript interpreter, in the same JavaScript environment. As we've seen, one frame can refer to any other frame using the `frames`, `parent`, and `top` properties. So, although JavaScript code in different frames is executed with different scope chains, the code in one frame can still refer to and use the variables and functions defined by code in another frame.

For example, suppose code in frame A defines a variable `i`:

```
var i = 3;
```

That variable is nothing more than a property of the global object -- a property of the Window object. Code in frame A could refer to the variable explicitly as such a property with either of these two expressions:

```
window.i  
self.i
```

Now suppose that frame A has a sibling frame B that wants to set the value of the variable `i` defined by the code in frame A. If frame B just sets a variable `i`, it merely succeeds in creating a new property of its own Window object. So instead, it must explicitly refer to the property `i` in its sibling frame with code like this:

```
parent.frames[0].i = 4;
```

Recall that the `function` keyword that defines functions declares a variable just like the `var` keyword does. If JavaScript code in frame A declares a function `f`, that function is defined only within frame A. Code in frame A can invoke `f` like this:

```
f( );
```

Code in frame B, however, must refer to `f` as a property of the Window object of frame A:

```
parent.frames[0].f( );
```

If the code in frame B needs to use this function frequently, it might assign the function to a variable of frame B so that it can more conveniently refer to the function:


```
var f = parent.frames[0].f;
```

Now code in frame B can invoke the function as `f()`, just as code in frame A does.

When you share functions between frames or windows like this, it is very important to keep the rules of lexical scoping in mind. A function is executed in the scope in which it was defined, not in the scope from which it is invoked. Thus, to continue with the previous example, if the function `f` refers to global variables, these variables are looked up as properties of frame A, even when the function is invoked from frame B.

If you don't pay careful attention to this, you can end up with programs that behave in unexpected and confusing ways. For example, suppose you define the following function in the `<head>` section of a multiframe document, with the idea that it will help with debugging:

```
function debug(msg) {
    alert("Debugging message from frame: " + name + "\n" + msg);
}
```

The JavaScript code in each of your frames can refer to this function as `top.debug()`. Whenever this function is invoked, however, it looks up the variable `name` in the context of the top-level window in which the function is defined, rather than the context of the frame from which it is invoked. Thus, the debugging messages always carry the name of the top-level window, rather than the name of the frame that sent the message, as was intended.

Remember that constructors are also functions, so when you define a class of objects with a constructor function and an associated prototype object, that class is defined only for a single window. Recall the `Complex` class we defined in [Chapter 8](#), and consider the following multiframed HTML document:

```
<head>
<script src="Complex.js"></script>
</head>
<frameset rows="50%,50%">
  <frame name="frame1" src="frame1.html">
  <frame name="frame2" src="frame2.html">
</frameset>
```

JavaScript code in the files `frame1.html` and `frame2.html` cannot create a `Complex` object with an expression like this:

```
var c = new Complex(1,2); // Won't work from either frame
```

Instead, code in these files must explicitly refer to the constructor function:

```
var c = new top.Complex(3,4);
```

Alternatively, code in either frame can define its own variable to refer more conveniently to the constructor function:

```
var Complex = top.Complex;
var c = new Complex(1,2);
```

Unlike user-defined constructors, predefined constructors are automatically predefined in all windows. Note, however, that each window has an independent copy of the constructor and an independent copy of the constructor's prototype object. For example, each window has its own copy of the `String()` constructor and the `String.prototype` object. So, if you write a new method for manipulating JavaScript strings and then make it a method of the `String` class by assigning it to the `String.prototype` object in the current window, all strings in that window can use the new method. However, the new method is not accessible to strings defined in other windows. Note that it does not matter which window holds a reference to the string; only the window in which the string was actually created matters.

13.11.4 Example: Colored Frames

[Example 13-7](#), a frame set that defines a grid of nine frames, demonstrates some of the techniques we've discussed in this chapter. The `<head>` section of the frame set includes a `<script>` that defines a JavaScript function named `setcolor()`. The `onload` event handler of the `<frameset>` tag invokes `setcolor()` once for each of the nine frames.

`setcolor()` is passed a `Window` object as its argument. It generates a random color and uses it with the `Document.write()` method to create a new document that is empty except for a background color. Finally, `setcolor()` uses the `setTimeout()` method to schedule itself to be called again in one second. This call to `setTimeout()` is the most interesting part of the example. Notice especially how it uses the `parent` and `name` properties of `Window` objects.

Example 13-7. A frame color animation

```
<head>
<title>Colored Frames</title>
<script>
function setcolor(w) {
    // Generate a random color
    var r = Math.floor((Math.random( ) * 256)).toString(16);
    var g = Math.floor((Math.random( ) * 256)).toString(16);
    var b = Math.floor((Math.random( ) * 256)).toString(16);
    var colorString = "#" + r + g + b;
```

```
// Set the frame background to the random color
w.document.write("<body bgcolor='" + colorString + "'></body>");
w.document.close( );

// Schedule another call to this method in one second.
// Since we call the setTimeout( ) method of the frame, the string
// will be executed in that context, so we must prefix properties
// of the top-level window with "parent.".
w.setTimeout('parent.setcolor(parent.' + w.name + ')', 1000);

// We could also have done the same thing more simply like this:
// setTimeout('setcolor(' + w.name + ')', 1000);
}
</script>
</head>
<frameset rows="33%,33%,34%" cols="33%,33%,34%"
  onload="for(var i = 0; i < 9; i++) setcolor(frames[i]);">
<frame name="f1" src="javascript:''"><frame name="f2"
src="javascript:''">
<frame name="f3" src="javascript:''"><frame name="f4"
src="javascript:''">
<frame name="f5" src="javascript:''"><frame name="f6"
src="javascript:''">
<frame name="f7" src="javascript:''"><frame name="f8"
src="javascript:''">
<frame name="f9" src="javascript:''">
</frameset>
```


Chapter 14. The Document Object

Every Window object has a `document` property. This property refers to a Document object that represents the HTML document displayed in the window. The Document object is probably the most commonly used object in client-side JavaScript. We've already seen several examples in this book that use the `write()` method of the Document object to insert dynamic content into a document while it is being parsed. In addition to the frequently used `write()` method, the Document object defines properties that provide information about the document as a whole: its URL, its last-modified date, the URL of the document that linked to it, the colors in which it is displayed, and so on.

Client-side JavaScript exists to turn static HTML documents into interactive programs -- it is the Document object that gives JavaScript interactive access to the content of otherwise static documents. In addition to the properties that provide information about a document as a whole, the Document object has a number of very important properties that provide information about document content. The `forms[]` array, for instance, contains Form objects that represent all the HTML forms in the document. And the `images[]` and `applets[]` arrays contain objects that represent the images and applets in the document. These arrays and the objects they contain open up a world of possibilities for client-side JavaScript programs, and the bulk of this chapter is devoted to documenting them.

This chapter covers the core features of the Document object that are implemented by virtually every JavaScript-enabled browser. Newer browsers, such as IE 4 and later and Netscape 6 and later, implement a full document object model, or DOM, that gives JavaScript complete access to and control over all document content. These advanced DOM features are covered in [Chapter 17](#).

14.1 Document Overview

To illustrate the scope and importance of the Document object, this chapter begins with a quick summary of the methods and properties of the object. The following sections also explain other important material that is important to understand before reading the rest of the chapter.

14.1.1 Document Methods

The Document object defines four key methods. One is the `write()` method, which we've already seen several times, and the other three are related:

```
close( )
```

Close or end a document that was begun with `open()`.

`open()`

Begin a new document, erasing any existing document content.

`write()`

Append text to the currently open document.

`writeln()`

Output text into the currently open document, and append a newline character.

14.1.2 Document Properties

The Document object defines the following properties:

`alinkColor`, `linkColor`, `vlinkColor`

These properties describe the colors of hyperlinks. `linkColor` is the normal color of an unvisited link. `vlinkColor` is the normal color of a visited link. `alinkColor` is the color of a link while it is activated (i.e., while the user is clicking on it). These properties correspond to the `alink`, `link`, and `vlink` attributes of the `<body>` tag.

`anchors[]`

An array of Anchor objects that represent the anchors in the document.

`applets[]`

An array of Applet objects that represent the Java applets in the document.

`bgColor`, `fgColor`

The background and foreground (i.e., text) colors of the document. These properties correspond to the `bgcolor` and `text` attributes of the `<body>` tag.

`cookie`

A special property that allows JavaScript programs to read and write HTTP cookies. See [Chapter 16](#) for details.

`domain`

A property that allows mutually trusted web servers within the same Internet domain to collaboratively relax certain security restrictions on interactions between their web pages. See [Chapter 21](#).

`forms[]`

An array of Form objects that represent the `<form>` elements in the document.

`images[]`

An array of Image objects that represent the `` elements in the document.

`lastModified`

A string that contains the modification date of the document.

`links[]`

An array of Link objects that represent the hypertext links in the document.

`location`

A deprecated synonym for the `URL` property.

`referrer`

The URL of the document containing the link that brought the browser to the current document, if any.

`title`

The text between the `<title>` and `</title>` tags for this document.

URL

A string specifying the URL from which the document was loaded. The value of this property is the same as the `location.href` property of the Window object, except when a server redirect has occurred.

14.1.3 The Document Object and Standards

The Document object and the set of elements (such as forms, images, and links) that it exposes to JavaScript programs form a document object model. Historically, different browser vendors have implemented different DOMs, which has made it difficult for JavaScript programmers to portably use the advanced features of the vendor-specific DOMs. Fortunately, the World Wide Web Consortium (or W3C; see <http://www.w3.org>) has standardized a DOM and issued two versions of this standard, known as Level 1 and Level 2. Recent browsers, such as Netscape 6 and later and IE 5 and later, implement some or most of these standards. See [Chapter 17](#) for all the details.

The DOM described in this chapter predates the W3C standards. By virtue of its nearly universal implementation, however, it is a de facto standard and is often referred to as the Level 0 DOM. You can use the techniques described in this chapter in any JavaScript-enabled web browser, with the exception of very old ones such as Netscape 2. Furthermore, the Document object methods and properties listed previously have been formalized as part of the Level 1 DOM, so they are guaranteed to remain supported by future browsers.

One important thing to understand about the W3C DOM standard is that it is a document object model for both XML and HTML documents. In this standard, the Document object provides generic functionality of use for both types of documents. HTML-specific functionality is provided by the `HTMLDocument` subclass. All the Document properties and methods described in this chapter are HTML-specific, and you can find more details about them under the "Document" entry in the client-side reference section of this book. You'll also find related information in the DOM reference section, under "Document" and "HTMLDocument."

14.1.4 Naming Document Objects

Before we begin our discussion of the Document object and the various objects it exposes, there is one general principle that you'll find it helpful to keep in mind. As you'll see, every `<form>` element in an HTML document creates a numbered element in the `forms[]` array of the Document object. Similarly, every `` element creates an element in the `images[]` array. The same applies for `<a>` and `<applet>` tags, which define elements in the `links[]` and `applets[]` arrays.

In addition to these arrays, however, a Form, Image, or Applet object may be referred to by name if its corresponding HTML tag is given a `name` attribute. When this attribute is

present, its value is used to expose the corresponding object as a property of the Document object. So, for example, suppose an HTML document contains the following form:

```
<form name="f1">
<input type="button" value="Push Me">
</form>
```

Assuming that the `<form>` is the first one in the document, your JavaScript code can refer to the resulting Form object with either of the following two expressions:

```
document.forms[0] // Refer to the form by position within the document
document.f1       // Refer to the form by name
```

In fact, setting the `name` attribute of a `<form>` also makes the Form object accessible as a named property of the `forms[]` array, so you could also refer to the form with either of these two expressions:

```
document.forms.f1 // Use property syntax
document.forms["f1"] // Use array syntax
```

The same applies for images and applets: using the `name` attribute in your HTML allows you to refer to these objects by name in your JavaScript code.

As you might imagine, it is convenient to give names to frequently used Document objects so that you can refer to them more easily in your scripts. We'll see this technique used a number of times in this and later chapters.

14.1.5 Document Objects and Event Handlers

To be interactive, an HTML document and the elements within it must respond to user events. We discussed events and event handlers briefly in [Chapter 12](#), and we've seen several examples that use simple event handlers. We'll see many more examples of event handlers in this chapter, because they are key to working with Document objects.

Unfortunately, we must defer a complete discussion of events and event handlers until [Chapter 19](#). For now, remember that event handlers are defined by attributes of HTML elements, such as `onclick` and `onmouseover`. The values of these attributes should be strings of JavaScript code. This code is executed whenever the specified event occurs on the HTML element.

In addition, there is one other way to define event handlers that we'll occasionally see used in this and later chapters. We'll see in this chapter that Document objects such as Form and Image objects have JavaScript properties that match the HTML attributes of

the `<form>` and `` tags. For example, the HTML `` tag has `src` and `width` attributes, and the JavaScript Image object has corresponding `src` and `width` properties. The same is true for event handlers. The HTML `<a>` tag supports an `onclick` event handler, for example, and the JavaScript Link object that represents a hyperlink has a corresponding `onclick` property. As another example, consider the `onsubmit` attribute of the `<form>` element. In JavaScript, the Form object has a corresponding `onsubmit` property. Remember that HTML is not case-sensitive, and attributes can be written in uppercase, lowercase, or mixed-case. In JavaScript, all event handler properties must be written in lowercase.

In HTML, event handlers are defined by assigning a string of JavaScript code to an event handler attribute. In JavaScript, however, they are defined by assigning a function to an event handler property. Consider the following `<form>` and its `onsubmit` event handler:

```
<form name="myform" onsubmit="return validateform( );">...</form>
```

In JavaScript, instead of using a string of JavaScript code that invokes a function and returns its result, we could simply assign the function directly to the event handler property like this:

```
document.myform.onsubmit = validateform;
```

Note that there are no parentheses after the function name. That is because we don't want to invoke the function here; we just want to assign a reference to it. As another example, consider the following `<a>` tag and its `onmouseover` event handler:

```
<a href="help.html" onmouseover="status='Get Help Now!';">Help</a>
```

If we happen to know that this `<a>` tag is the first one in the document, we can refer to the corresponding Link object as `document.links[0]` and set the event handler this way instead:

```
document.links[0].onmouseover = function( ) { status = 'Get Help Now!'; }
```

See [Chapter 19](#) for a complete discussion of assigning event handlers in this way.

14.2 Dynamically Generated Documents

One of the most important features of the Document object (and perhaps of client-side JavaScript in general) is the `write()` method, which allows you to dynamically generate web-page content from your JavaScript programs. This method can be used in

two ways. The first and simplest way to use it is within a script, to output dynamically generated HTML into the document that is currently being parsed. This was discussed in [Chapter 12](#). Consider the following code, which uses `write()` to add the current date and the document's last-modified date to an otherwise static HTML document:

```
<script>
var today = new Date( );
document.write("<p>Document accessed on: " + today.toString( ));
document.write("<br>Document modified on: " + document.lastModified);
</script>
```

Using the `write()` method in this way is an extremely common JavaScript programming technique, and you'll see it in many scripts.

Be aware, however, that you can use the `write()` method to output HTML to the current document only while that document is being parsed. That is, you can call `document.write()` from within `<script>` tags only because these scripts are executed as part of the document parsing process. In particular, if you call `document.write()` from within an event handler and that handler is invoked once the document has already been parsed, you will end up overwriting the entire document (including its event handlers), instead of appending text to it. The reason for this will become clear as we examine the second way to use the `write()` method.

In addition to adding dynamic content to the current document as it is being parsed, `write()` can be used in conjunction with the `open()` and `close()` Document methods to create entirely new documents within a window or frame. Although you cannot usefully write to the current document from an event handler, there is no reason why you can't write to a document in another window or frame; doing so can be a useful technique with multiwindow or multiframe web sites. For example, JavaScript code in one frame of a multiframe site might display a message in another frame with code like this:

```
<script>
// Start a new document, erasing any content that was already in
frames[0]
parent.frames[0].document.open( );
// Add some content to the document
parent.frames[0].document.write("<hr>Hello from your sibling
frame!<hr>");
// And close the document when we're done
parent.frames[0].document.close( );
</script>
```

To create a new document, we first call the `open()` method of the Document object, then call `write()` any number of times to output the contents of the document, and finally call the `close()` method of the Document object to indicate that we have finished. This last step is important; if you forget to close the document, the browser does

not stop the document loading animation it displays. Also, the browser may buffer the HTML you have written; it is not required to display the buffered output until you explicitly end the document by calling `close()`.

In contrast to the `close()` call, which is required, the `open()` call is optional. If you call the `write()` method on a document that has already been closed, JavaScript implicitly opens a new HTML document, as if you had called the `open()` method. This explains what happens when you call `document.write()` from an event handler within the same document -- JavaScript opens a new document. In the process, however, the current document (and its contents, including scripts and event handlers) is discarded. This is never what you want to do, and it can even cause some early browsers (such as Netscape 2) to crash. As a general rule of thumb, a document should never call `write()` on itself from within an event handler.

A couple of final notes about the `write()` method. First, many people do not realize that the `write()` method can take more than one argument. When you pass multiple arguments, they are output one after another, just as if they had been concatenated. So instead of writing:

```
document.write("Hello, " + username + " Welcome to my home page!");
```

you might equivalently write:

```
var greeting = "Hello, ";  
var welcome = " Welcome to my home page!";  
document.write(greeting, username, welcome);
```

The second point to note about the `write()` method is that the Document object also supports a `writeln()` method, which is identical to the `write()` method in every way except that it appends a newline after outputting its arguments. Since HTML ignores line breaks, this newline character usually doesn't make a difference, but as we'll see in a bit, the `writeln()` method can be convenient when you're working with non-HTML documents.

[Example 14-1](#) shows how you might create a complex dialog box with the Window `open()` method and the methods of the Document object. This example registers an `onerror` event handler function for the window; the function is invoked when a JavaScript error occurs. The error handler function creates a new window and uses the Document object methods to create an HTML form within the window. The form allows the user to see details about the error that occurred and email a bug report to the author of the JavaScript code.

[Figure 14-1](#) shows a sample window. Recall from the discussion of the `onerror` error handler in [Chapter 13](#) that Netscape 6 does not pass the correct arguments to the error

handler function. For this reason, the output on Netscape 6 does not match what is illustrated here.

Figure 14-1. Using a browser window as a dialog box



Example 14-1. Dynamically creating a dialog window

```
<script>
// A variable we use to ensure that each error window we create is
unique
var error_count = 0;

// Set this variable to your email address
var email = "myname@mydomain.com";

// Define the error handler. It generates an HTML form so the user
// can report the error to the author.
function report_error(msg, url, line)
{
    var w = window.open("", // URL (none specified)
                        "error"+error_count++, // Name (force it to be
unique)
                        "resizable,status,width=625,height=400"); //
Features
    // Get the Document object of the new window
    var d = w.document;

    // Output an HTML document, including a form, into the new window
    // Note that we omit the optional call to document.open( )
    d.write('<div align="center">');
```

```

    d.write('<font size="7" face="helvetica"><b>');
    d.write('OOPS.... A JavaScript Error Has Occurred!');
    d.write('</b></font><br><hr size="4" width="80%">');
    d.write('<form action="mailto:' + email + '" method=post');
    d.write(' enctype="text/plain">');
    d.write('<font size="3">');
    d.write('<i>Click the "Report Error" button to send a bug
report.</i><br>');
    d.write('<input type="submit" value="Report Error">&nbsp;&nbsp;&nbsp;');
    d.write('<input type="button" value="Dismiss" onclick="self.close(
);">');
    d.write('</div><div align="right">');
    d.write('<br>Your name <i>(optional)</i>: ');
    d.write('<input size="42" name="name" value="">');
    d.write('<br>Error Message: ');
    d.write('<input size="42" name="message" value="' + msg + '">');
    d.write('<br>Document: <input size="42" name="url" value="' + url +
'">');
    d.write('<br>Line Number: <input size="42" name="line"
value="' + line + '">');
    d.write('<br>Browser Version: ');
    d.write('<input size="42" name="version"
value="' + navigator.userAgent + '">');
    d.write('</div></font>');
    d.write('</form>');
    // Remember to close the document when we're done
    d.close( );

    // Return true from this error handler, so that JavaScript does not
    // display its own error dialog box
    return true;
}

// Before the event handler can take effect, we have to register it
// for a particular window
self.onerror = report_error;
</script>

<script>
// The following line of code purposely causes an error as a test
alert(no_such_variable);
</script>

```

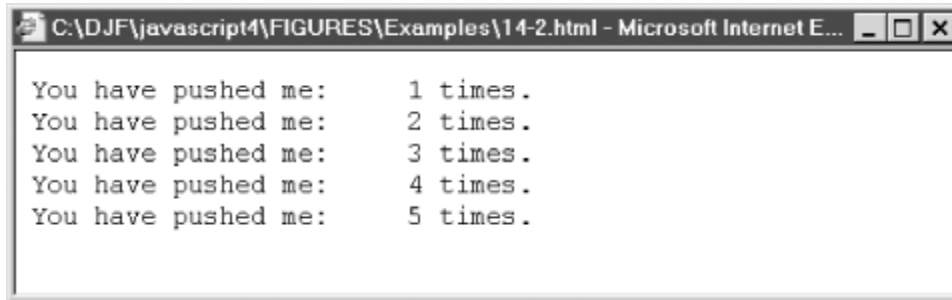
14.2.1 Non-HTML Documents

When you call the Document `open()` method with no arguments, it opens a new HTML document. Remember, though, that web browsers can display a number of other data formats besides HTML text. When you want to dynamically create and display a document using some other data format, you call the `open()` method with a single argument, which is the MIME type you desire.^[1]

^[1] This argument to the `open()` method has not been standardized by the W3C DOM. It works in IE 4 and later, and in Netscape 3 and 4. Surprisingly, it does not work in Netscape 6: only HTML documents are supported by that browser.

The MIME type for HTML is `text/html`. The most common format besides HTML is plain text, with a MIME type of `text/plain`. If you want to use the `write()` method to output text that uses newlines, spaces, and Tab characters for formatting, you should open the document by passing the string "text/plain" to the `open()` method. [Example 14-2](#) shows one way you might do this. It implements a `debug()` function that you can use to output plain-text debugging messages from your scripts into a separate window that appears when needed. [Figure 14-2](#) shows what the resulting window looks like.

Figure 14-2. A window for plain-text debugging output



Example 14-2. Creating a plain-text document

```
<script>
var _console = null;

function debug(msg)
{
    // Open a window the first time we are called, or after an existing
    // console window has been closed
    if ((_console == null) || (_console.closed)) {
        _console =
window.open("", "console", "width=600,height=300,resizable");
        // Open a document in the window to display plain text
        _console.document.open("text/plain");
    }

    _console.focus( );           // Make the window visible
    _console.document.writeln(msg); // Output the message to it
    // Note that we purposely do not call close( ). Leaving the
    // document open allows us to append to it later.
}
</script>

<!-- Here's an example of using this script -->
<script>var n = 0;</script>
<form>
<input type="button" value="Push Me"
        onclick="debug('You have pushed me:\t' + ++n + ' times.');" />
</form>
```

14.3 Document Color Properties

The `bgColor`, `fgColor`, `linkColor`, `alinkColor`, and `vlinkColor` properties of the Document object specify foreground, background, and link colors for the document. They are read/write properties, but they can be set only before the `<body>` tag is parsed. You can set them dynamically with JavaScript code in the `<head>` section of a document, or you can set them statically as attributes of the `<body>` tag, but you cannot set them elsewhere. The exception to this rule is the `bgColor` property. In many browsers, you can set this property at any time; doing so causes the background color of the browser window to change.^[2] Other than `bgColor`, the color properties of the Document object merely expose attributes of the `<body>` tag and are basically uninteresting.

^[2] There is a bug in Netscape 3 on Unix platforms such that changing the background color can make the contents of the page disappear (usually until the window is scrolled or otherwise redrawn). In Netscape 6, you can set the `bgColor` only once; any additional settings are ignored.

Each of these color properties has a string value. To set a color, you can use one of the predefined HTML color names, or you can specify the color as red, green, and blue color values, expressed as a string of six hexadecimal digits in the form `#RRGGBB`. You may recall that [Example 13-7](#) set the `bgcolor` attribute of the `<body>` tag to a color string expressed in this fashion.

In the W3C DOM standard, the color properties of the Document object are deprecated in favor of properties of the Element object that represents the `<body>` tag. Furthermore, the HTML 4 standard deprecates the color attributes of the `<body>` tag in favor of CSS style sheets. What this means is that you probably should not write scripts that rely heavily on these doubly deprecated color properties!

14.4 Document Information Properties

Several properties of the Document object provide information about the document as a whole. For example, the following code shows how you can use the `lastModified`, `title`, and `URL` properties to include an automatic timestamp within a document. This feature allows users to judge how up-to-date (or out-of-date) a document is, and it can also be useful information when a document is printed.

```
<hr><font size="1">  
Document: <i><script>document.write(document.title);</script></i><br>  
URL: <i><script>document.write(document.URL);</script></i><br>  
Last Update:  
<i><script>document.write(document.lastModified);</script></i>  
</font>
```

`referrer` is another interesting property: it contains the URL of the document from which the user linked to the current document. One possible use is to save this value in a hidden field of a form on your web page. When the user submits the form (for whatever reason your page contains the form in the first place), you can save the referrer data on the server so you can analyze the links that refer to your page and track the percentage of

hits that come through various links. Another use of this property is a trick to prevent unauthorized links to your page from working correctly. For example, suppose you want to allow other sites to link only to the top-level page on your site. You can use the `referrer` property in conjunction with the `location` property of the Window object to redirect any links from outside the site to the top-level home page:

```
<script>
// If linked from somewhere offsite, go to home page first
if (document.referrer == "" || document.referrer.indexOf("mysite.com")
== -1)
    window.location = "http://home.mysite.com";
</script>
```

Don't consider this trick to be any kind of serious security measure, of course. One obvious flaw is that it doesn't work for browsers that don't support JavaScript or for users who have disabled JavaScript.

14.5 Forms

The `forms[]` array of the Document object contains Form objects that represent any `<form>` elements in the document. Because HTML forms contain push buttons, text input fields, and the other input elements that usually comprise the GUI of a web application, the Form object is very important in client-side JavaScript. The Form object has an `elements[]` property that contains objects that represent the HTML input elements contained within the form. These Element objects allow JavaScript programs to set default values in the form and to read the user's input from the form. They are also important sites for the event handlers that add interactivity to a program.

Because forms and their elements are such a large and important part of client-side JavaScript programming, they deserve a chapter of their own. We will return to the `forms[]` array and the Form object in [Chapter 15](#).

14.6 Images

The `images[]` property of the Document object is an array of Image elements, each representing one of the inline images, created with an `` tag, that is contained in the document. The `images[]` array and the Image object were added in JavaScript 1.1. While web browsers have always been able to display images with the `` tag, the addition of the Image object was a major step forward -- it allowed programs to dynamically manipulate those images.

14.6.1 Image Replacement with the `src` Property

The main feature of the Image object is that its `src` property is read/write. You can read this property to obtain the URL from which an image was loaded, and, more importantly,

you can set the `src` property to make the browser load and display a new image in the same space. For this to work, the new image must have the same width and height as the original one.

In practice, the most common use for image replacement is to implement image rollovers, in which an image changes when the mouse pointer moves over it. When you make images clickable by placing them inside your hyperlinks, rollover effects are a powerful way to invite the user to click on the image. Here is a simple HTML fragment that displays an image within an `<a>` tag and uses JavaScript code in the `onmouseover` and `onmouseout` event handlers to create a rollover effect:

```
<a href="help.html"
  onmouseover="document.helpimage.src='images/help_rollover.gif';"
  onmouseout="document.helpimage.src='images/help.gif';">

</a>
```

Note that in this code fragment we gave the `` tag a `name` attribute, to make it easy to refer to the corresponding Image object in the event handlers of the `<a>` tag. We used the `border` attribute to prevent the browser from displaying a blue hyperlink border around the image. The event handlers of the `<a>` tag do all the work: they change the image that is displayed simply by setting the `src` property of the image to the URLs of the desired images.

The ability to dynamically replace one image in a static HTML document with another image opens the door to any number of special effects, from animation to digital clocks that update themselves in real time. With a bit of thought, you can probably imagine many more potential uses for this technique.

14.6.2 Offscreen Images and Caching

To make image-replacement techniques viable, the animations or other special effects need to be responsive. This means that we need some way to ensure that the necessary images are "pre-fetched" into the browser's cache. To force an image to be cached, we first create an offscreen image using the `Image()` constructor. Next, we load an image into it by setting its `src` property to the desired URL (exactly as we would do for an onscreen image). Later, when the same URL is used for an onscreen image, we know it can be quickly loaded from the browser's cache, rather than slowly loaded over the network. Note that we never actually do anything with the offscreen image we create. In particular, we do *not* assign the offscreen Image object into the `images[]` array of the document.

The image-rollover code fragment shown in the previous section did not pre-fetch the rollover image it used, so the user will probably notice a delay in the rollover effect the

first time she moves the mouse over the image. To fix this problem, we could modify the code as follows.

```
<script>
// Create an offscreen image and pre-fetch the rollover image.
// Note that we don't bother saving a reference to the offscreen image,
// since there is nothing we can do with it later.
(new Image(80,20)).src = "images/help_rollover.gif";
</script>
<a href="help.html"
    onmouseover="document.helpimage.src='images/help_rollover.gif';"
    onmouseout="document.helpimage.src='images/help.gif';">

</a>
```

[Example 14-3](#) shows code that performs a simple animation using image replacement and uses offscreen images to pre-fetch the frames of the animation. Note that in this example we retain the offscreen image objects we create, because they are a convenient way to hold the URLs of the images that make up the animation. To perform the animation, we assign the `src` property of one of the offscreen images to the `src` property of the onscreen image that is the subject of the animation.

Example 14-3. An animation using image replacement

```
<!-- The image that will be animated. Give it a name for convenience. -
->


<script>
// Create a bunch of offscreen images, and pre-fetch the "frames"
// of the animation into them so that they're cached when we need them
var aniframes = new Array(10);
for(var i = 0; i < 10; i++) {
    aniframes[i] = new Image(    );           // Create an
offscreen image
    aniframes[i].src = "images/" + i + ".gif"; // Tell it what URL
to load
}

var frame = 0;           // The frame counter: keeps track of current
frame
var timeout_id = null; // Allows us to stop the animation with
clearTimeout(    )

// This function performs the animation. Call it once to start.
// Note that we refer to the onscreen image using its name attribute.
function animate(    ) {
    document.animation.src = aniframes[frame].src; // Display the
current frame
    frame = (frame + 1)%10;           // Update the
frame counter
    timeout_id = setTimeout("animate(    )", 250); // Display the
next frame later
```

```

}
</script>

<form> <!-- This form contains buttons to control the animation -->
  <input type="button" value="Start"
    onclick="if (timeout_id == null) animate( );">
  <input type="button" value="Stop"
    onclick="if (timeout_id) clearTimeout(timeout_id);
timeout_id=null;">
</form>

```

14.6.3 Image Event Handlers

In [Example 14-3](#), our animation does not begin until the user clicks the **Start** button, which allows plenty of time for our images to be loaded into the cache. But what about the more common case in which we want to automatically begin an animation as soon as all the necessary images are loaded? It turns out that images, whether created onscreen with an `` tag or offscreen with the `Image()` constructor, have an `onload` event handler that is invoked when the image is fully loaded.

The following code fragment shows how we could modify [Example 14-3](#) to use this event handler to count the number of images that have loaded and automatically start the animation when all the images have loaded. Since offscreen images are not part of the HTML document, the event handler cannot be assigned as an HTML attribute. Instead, we simply assign a function to the `onload` property of each `Image` object we create. When each image is loaded, the browser calls the function.

```

var aniframes = new Array(10); // Hold the offscreen animation frames.
var num_loaded_images = 0;     // How many have been loaded so far?

// This function is used as an event handler. It counts how many images
// have been loaded and, when all have been loaded, it starts the
// animation.
function countImages( ) {
    if (++num_loaded_images == aniframes.length) animate( );
}

// Create the offscreen images and assign the image URLs.
// Also assign an event handler to each image so we can track how many
// images
// have been loaded. Note that we assign the handler before the URL,
// because
// otherwise the image might finish loading (e.g., if it is already
// cached)
// before we assign the handler, and then the handler would never be
// triggered.
for(var i = 0; i < 10; i++) {
    aniframes[i] = new Image( ); // Create an
    // offscreen image
    aniframes[i].onload = countImages; // Assign the event
    // handler
}

```

```
    aniframes[i].src = "images/" + i + ".gif";    // Tell it what URL
to load
}
```

In addition to the `onload` event handler, the Image object supports two others. The `onerror` event handler is invoked when an error occurs during image loading, such as when the specified URL refers to corrupt image data. The `onabort` handler is invoked if the user cancels the image load (for example, by clicking the **Stop** button in the browser) before it has finished. For any image, one (and only one) of these handlers is called.

In addition to these handlers, each Image object also has a `complete` property. This property is `false` while the image is loading; it is changed to `true` once the image has loaded or once the browser has stopped trying to load it. In other words, the `complete` property becomes `true` after one of the three possible event handlers is invoked.

14.6.4 Other Image Properties

The Image object has a few other properties as well. Most of them are simply mirror attributes of the `` tag that created the image. The `width`, `height`, `border`, `hspace`, and `vspace` properties are integers that specify the size of the image, the width of its border, and the size of its horizontal and vertical margins. These properties are set by the attributes of the `` tag that share their names. In Netscape 3 and 4, the properties are read-only, but in IE 4 and later and Netscape 6 and later, you can also assign values to these properties to dynamically change the size, border, or margins of the image.

The `lowsrc` property of the Image object mirrors the `lowsrc` attribute of the `` tag. It specifies the URL of an optional image to display when the page is viewed on a low-resolution device. The `lowsrc` property is a read/write string, like `src`, but unlike the `src` property, setting `lowsrc` does not cause the browser to load and display the newly specified, low-resolution image. If you want to perform an animation or some other special effect that works with low-resolution images as well as high-resolution ones, always remember to update the `lowsrc` property before you set the `src` property. If the browser is running on a low-resolution device when you set the `src` literal, it loads the new `lowsrc` image instead.

14.6.5 Image-Replacement Example

Because image replacement is such a versatile technique, we'll end our discussion of the Image object with an extended example. [Example 14-4](#) defines a `ToggleButton` class that uses image replacement to simulate a graphical checkbox. Because this class uses images that we provide, we can use bolder graphics than those plain old graphics used by the standard HTML Checkbox object. [Figure 14-3](#) shows how these toggle-button graphics could appear on a web page. This is a complex, real-world example and is worth studying carefully.

Figure 14-3. ToggleButtons implemented with image replacement



Example 14-4. Implementing a ToggleButton with image replacement

```
<script language="JavaScript1.1">
// This is the constructor function for our new ToggleButton class.
// Calling it creates a ToggleButton object and outputs the required
// <a> and <img> tags into the specified document at the current
location.
// Therefore, don't call it for the current document from an event
handler.
// Arguments:
//   document: The Document object in which the buttons are to be
created.
//   checked: A boolean that says whether the button is initially
checked.
//   label: An optional string that specifies text to appear after
the button.
//   onclick: An optional function to be called when the toggle
button is
//             clicked. It is passed a boolean indicating the new
state of
//             the button. You can also pass a string, which is
converted
//             to a function that is passed a boolean argument named
"state".
function ToggleButton(document, checked, label, onclick)
{
    // The first time we are called (and only the first time), we have
    // to do some special stuff. First, now that the prototype object
    // is created, we can set up our methods.
    // Second, we need to load the images we'll be using.
    // Doing this gets the images in the cache for when we need them.
    if (!ToggleButton.prototype.over) {
        // Initialize the prototype object to create our methods
        ToggleButton.prototype.over = _ToggleButton_over;
        ToggleButton.prototype.out = _ToggleButton_out;
        ToggleButton.prototype.click = _ToggleButton_click;

        // Now create an array of Image objects and assign URLs to
        them.
        // The URLs of the images are configurable and are stored in an
        // array property of the constructor function itself. They are
```

```

        // initialized below. Because of a bug in Netscape, we have
        // to maintain references to these images, so we store the
array
        // in a property of the constructor rather than using a local
variable.
        ToggleButton.images = new Array(4);
        for(var i = 0; i < 4; i++) {
            ToggleButton.images[i] = new Image(ToggleButton.width,
                                                ToggleButton.height);
            ToggleButton.images[i].src = ToggleButton.imagenames[i];
        }
    }

    // Save some of the arguments we were passed
    this.document = document;
    this.checked = checked;

    // Remember that the mouse is not currently on top of us
    this.highlighted = false;

    // Save the onclick argument to be called when the button is
clicked.
    // If it is not already a function, attempt to convert it
    // to a function that is passed a single argument, named "state".
    this.onclick = onclick;
    if (typeof this.onclick == "string")
        this.onclick = new Function("state", this.onclick);

    // Figure out what entry in the document.images[] array the images
    // for this checkbox will be stored in
    var index = document.images.length;

    // Now output the HTML code for this checkbox. Use <a> and <img>
tags.
    // The event handlers we output here are confusing but crucial to
the
    // operation of this class. The "_tb" property is defined below, as
    // are the over(), out(), and click( ) methods.
    document.write('&nbsp;<a href="about:blank" ' +
        'onmouseover="document.images[' + index + ']._tb.over( );return
true;" '+
        'onmouseout="document.images[' + index + ']._tb.out( )" '+
        'onclick="document.images[' + index + ']._tb.click( ); return
false;">');
    document.write('');
    if (label) document.write(label);
    document.write('</a></br>');

    // Now that we've output the <img> tag, save a reference to the
    // Image object that it created in the ToggleButton object
    this.image = document.images[index];

```

```

        // Also make a link in the other direction, from the Image object
        // to this ToggleButton object. Do this by defining a "_tb"
property
        // in the Image object.
        this.image._tb = this;
    }

// This becomes the over( ) method
function _ToggleButton_over( )
{
    // Change the image, and remember that we're highlighted
    this.image.src = ToggleButton.imagenames[this.checked + 2];
    this.highlighted = true;
}

// This becomes the out( ) method
function _ToggleButton_out( )
{
    // Change the image, and remember that we're not highlighted
    this.image.src = ToggleButton.imagenames[this.checked + 0];
    this.highlighted = false;
}

// This becomes the click( ) method
function _ToggleButton_click( )
{
    // Toggle the state of the button, change the image, and call the
    // onclick method, if it was specified for this ToggleButton
    this.checked = !this.checked;
    this.image.src =
ToggleButton.imagenames[this.checked+this.highlighted*2];
    if (this.onclick) this.onclick(this.checked);
}

// Initialize static class properties that describe the checkbox
images. These
// are just defaults. Programs can override them by assigning new
values.
// But they should be overridden *before* any ToggleButtons are
created.
ToggleButton.imagenames = new Array(4);           // Create an array
ToggleButton.imagenames[0] = "images/button0.gif"; // The unchecked
box
ToggleButton.imagenames[1] = "images/button1.gif"; // The box with a
checkmark
ToggleButton.imagenames[2] = "images/button2.gif"; // Unchecked but
highlighted
ToggleButton.imagenames[3] = "images/button3.gif"; // Checked and
highlighted
ToggleButton.width = ToggleButton.height = 25;    // Size of all
images
</script>

<!-- Here's how we might use the ToggleButton class -->
Optional extras:<br>
<script language="JavaScript1.1">
// Create ToggleButton objects and output the HTML that implements them

```



```

// One button has no click handler, one has a function, and one has a
string
var tb1 = new ToggleButton(document, true, "56K Modem");
var tb2 = new ToggleButton(document, false, "Laser Printer",
                           function(clicked) {alert("printer: " +
clicked);});
var tb3 = new ToggleButton(document, false, "Tape Backup Unit",
                           "alert('Tape backup: ' + state)");
</script>

<!-- Here's how we can use the ToggleButton objects from event handlers
-->
<form>
<input type="button" value="Report Button States"
       onclick="alert(tb1.checked + '\n' + tb2.checked + '\n' +
tb3.checked)">
<input type="button" value="Reset Buttons"
       onclick="if (tb1.checked) tb1.click( );
               if (tb2.checked) tb2.click( );
               if (tb3.checked) tb3.click( );">
</form>

```

14.7 Links

The `links[]` array of the Document object contains Link objects that represent each of the hypertext links in a document. Recall that HTML hypertext links are coded with the `href` attribute of the `<a>` tag. In JavaScript 1.1 and later, the `<area>` tag in a client-side image map also creates a Link object in the Document `links[]` array.

The Link object represents the URL of the hypertext link and contains all the properties that the Location object (introduced in [Chapter 13](#)) does. For example, the `href` property of a Link object contains the complete text of the URL to which it is linked, while the `hostname` property contains only the hostname portion of that URL. See the client-side reference section for a complete list of these URL-related properties.

[Example 14-5](#) shows a function that generates a list of all the links in a document. Note the use of the Document `write()` and `close()` methods to dynamically generate a document, as discussed earlier in this chapter.

Example 14-5. Listing the links in a document

```

/*
 * FILE: listlinks.js
 * List all links in the specified document in a new window
 */
function listlinks(d) {
    var newwin = window.open("", "linklist",
"menubar,scrollbars,resizable,width=600,height=300");

    for (var i = 0; i < d.links.length; i++) {
        newwin.document.write('<a href="' + d.links[i].href + '>')

```

```
        newwin.document.write(d.links[i].href);
        newwin.document.writeln("</a><br>");
    }
    newwin.document.close( );
}
```

14.7.1 Links, Web Crawlers, and JavaScript Security

One obvious use of the Link object and the `links[]` array is to write a web-crawler program. This program runs in one browser window or frame and reads web pages into another window or frame (by setting the `location` property of the Window object). For each page it reads in, it looks through the `links[]` array and recursively follows them. If carefully written (so it doesn't get caught in infinite recursion or start going in circles), such a program can be used, for example, to generate a list of all web pages that are accessible from a given starting page. This list can be quite useful in web-site maintenance.

Don't expect to crawl the entire Internet using these techniques, however. For security reasons, JavaScript does not allow an unsigned script in one window or frame to read the properties (such as `document.links`) of another window or frame unless both windows are displaying documents that came from the same web server. This restriction prevents important security breaches. Imagine that an employee at a large, security-conscious company is browsing the Internet through a corporate firewall and is also using another browser window to browse proprietary company information on the corporate intranet. Without the security restriction we've described, an untrusted script from some random Internet site could snoop on what was going on in the other window. The authors of the snooping script might not be able to glean much useful information from the `links[]` array of the proprietary documents, but this would nevertheless be a serious breach of security.

The web-crawler program we have described is not a threat to Internet security or privacy, but unfortunately, it is still subject to the general security restrictions of JavaScript, which prevent it from crawling very far beyond the site from which it was loaded. (When the crawler loads a page from a different site, it appears as if that page simply has no links on it.) See [Chapter 21](#) for a complete discussion of JavaScript security, including a description of how to avoid this security restriction with signed scripts.

14.7.2 Link Event Handlers

The Link object supports a number of interesting event handlers. We already saw the `onmouseover` event handler in [Section 13.3](#), where it was used with both `<a>` and `<area>` tags to change the message in the browser's status line when the mouse moved over the link. The `onclick` event handler is invoked when the user clicks on a hypertext link. In JavaScript 1.1 and later, if this event handler returns `false`, the browser doesn't follow the link as it would otherwise. As of JavaScript 1.1, both the `<a>` and `<area>` tags support

an `onmouseout` event handler. This is simply the opposite of the `onmouseover` handler -- it is run when the mouse pointer moves off a hypertext link.

The event-handling model has become much more general in JavaScript 1.2, and links support quite a few other event handlers. See [Chapter 19](#) for details.

Finally, it is worth mentioning that `href` and the other URL properties of the Link object are read/write. Thus, you can write a JavaScript program that dynamically modifies the destinations of hypertext links! Here is a frivolous piece of JavaScript-enhanced HTML that uses a Link event handler to write to the `href` property and create a link whose destination is randomly chosen from the set of other links in the document:

```
<a href="about:"
  onmouseover="status = 'Take a chance... Click me.'; return true;"
  onclick="this.href =
            document.links[Math.floor(Math.random(
) * document.links.length)];"
>
Random Link
</a>
```

This example demonstrates all the features of the Link object that we've considered: the `links[]` array, the use of Link event handlers, and the dynamic setting of the destination of a link. Note that the example sets the `href` property of the link but doesn't bother to read the `href` property of the link it randomly chooses. Instead, it relies on the `toString()` method of the Link object to return the URL.

14.8 Anchors

The `anchors[]` array of the Document object contains Anchor objects representing named locations in the HTML document that are marked with the `<a>` tag and its `name` attribute. The `anchors[]` array has existed since JavaScript 1.0, but the Anchor object is new in JavaScript 1.2. In previous versions, the elements of the `anchors[]` array were all undefined, and only the `length` property was useful.

The Anchor object is a simple one. The only standard property it defines is `name`, which is the value of the HTML `name` attribute. As with the Link object, the text that appears between the `<a>` and `` tags of the anchor is specified by the `text` property in Netscape 4 and by the `innerText` property in Internet Explorer 4. Neither of these properties is supported by the W3C DOM standard, but we'll see other ways to obtain the text content of an element in [Chapter 17](#).

[Example 14-6](#) shows a function that creates a navigation window for a specified document. It displays the `text`, `innerText`, or `name` of all the anchors in the document. The anchor text or name is displayed within hypertext links -- clicking on any anchor causes the original window to scroll to display that anchor. The code in this example is

particularly useful if you write your HTML documents so that all section headings are enclosed in anchors. For example:

```
<a name="sect14.6"><h2>The Anchor Object</h2></a>
```

Example 14-6. Listing all anchors

```
/*
 * FILE: listanchors.js
 * The function listanchors( ) is passed a document as its argument
and opens
 * a new window to serve as a "navigation window" for that document.
The new
 * window displays a list of all anchors in the document. Clicking on
any
 * anchor in the list causes the document to scroll to the position of
that
 * anchor. A document should not call this function on itself until it
is
 * fully parsed, or at least until all the anchors in it are parsed.
 */
function listanchors(d) {
    // Open the new window
    var newwin = window.open("", "navwin",

"menubar=yes,scrollbars=yes,resizable=yes," +
                                "width=600,height=300");

    // Give it a title
    newwin.document.writeln("<h1>Navigation Window:<br>" +
                                document.title + "</h1>");

    // List all anchors
    for(var i = 0; i < d.anchors.length; i++) {
        // For each anchor object, determine the text to display.
        // First, try to get the text between <a> and </a> using a
        // browser-dependent property. If none, use the name instead.
        var a = d.anchors[i];
        var text = null;
        if (a.text) text = a.text; // Netscape
4
        else if (a.innerText) text = a.innerText; // IE 4+
        if ((text == null) || (text == '')) text = a.name; // Default

        // Now output that text as a link. Note the use of the location
        // property of the original window.
        newwin.document.write('<a href="#" + a.name + "' +
                                ' onclick="opener.location.hash="' +
a.name +
                                '"; return false;">');
        newwin.document.write(text);
        newwin.document.writeln('</a><br>');
    }
    newwin.document.close( ); // Never forget to close the document!
}
```

14.9 Applets

The `applets[]` array of the Document object contains objects that represent the applets embedded in the document with the `<applet>` or `<object>` tag. An applet is a portable, secure Java program that is loaded over the Internet and executed by the web browser; both Netscape and Internet Explorer support Java (although IE 6 no longer includes Java support by default).

As of Netscape 3 and Internet Explorer 3, both browsers allow JavaScript to invoke public methods and read and write the public properties of Java applets. (As we'll see in [Chapter 22](#), Netscape also supports much richer bidirectional interactions between JavaScript and Java.) All applets have a few standard public methods that they inherit from their superclasses, but the most interesting methods and properties vary on a case-by-case basis. If you are the author of the applet that you want to control from JavaScript, you already know what public methods and properties it defines. If you are not the author, you should consult the applet's documentation to determine what you can do with it.

Here's how you might embed a Java applet in a web page with the `<applet>` tag and then invoke the `start()` and `stop()` methods of that applet from JavaScript event handlers:

```
<applet name="animation" code="Animation.class" width="500"
height="200">
</applet>
<form>
<input type="button" value="Start" onclick="document.animation.start(
);">
<input type="button" value="Stop" onclick="document.animation.stop(
);">
</form>
```

All applets define `start()` and `stop()` methods. In this hypothetical example, the methods cause an animation to start and stop; by defining the HTML form, we've given the user control over starting and stopping the applet. Note that we've used the `name` attribute of the `<applet>` tag, so we can refer to the applet by name, rather than as a numbered element of the `applets[]` array.

This example does not fully demonstrate the power of JavaScript to script Java applets: the Java methods invoked from the JavaScript event handlers are passed no arguments and return no values. In fact, JavaScript can pass numbers, strings, and boolean values as arguments to Java methods and can accept numbers, strings, and boolean return values from those functions. (As we'll see in [Chapter 22](#), Netscape can also pass and return JavaScript and Java objects to and from Java methods.) The automatic conversion of data between JavaScript and Java allows for rich interactions between the two programming environments. For example, an applet might implement a method that returns a string of JavaScript code. JavaScript could then use the `eval()` method to evaluate that code.

Applets can also implement methods that don't operate on the applet itself, but instead simply serve as conduits between JavaScript and the Java environment. For instance, an applet might define a method that invokes the `System.getProperty()` method for a given string argument. This applet would allow JavaScript to look up the value of Java system properties and determine, for example, the version of Java that is supported by the browser.

14.10 Embedded Data

The `embeds[]` array contains objects that represent data (other than applets) embedded in the document with the `<embed>` or `<object>` tag. Embedded data can take many forms (audio, video, spreadsheets, etc.). The browser must have an appropriate viewer installed or available so that it can display the data to the user. In Netscape, special modules known as plugins are responsible for displaying embedded data. In Internet Explorer, embedded data is displayed by ActiveX controls. Both plugins and ActiveX controls can be automatically downloaded from the network and installed as needed.

While the elements of the `applets[]` array all represent Java applets, the elements of the `embeds[]` array tend to be more diverse, and few generalizations can be made about them. The properties and methods of these objects depend upon the particular plugin or ActiveX control that is used to display the embedded data. You should consult the vendor-specific documentation for the plugin or ActiveX control you are using. If it supports any kind of scripting from JavaScript, the documentation should say so, and it should describe the properties and methods that you can use from JavaScript. For example, the documentation for the LiveVideo plugin from Netscape says that the LiveVideo object in the `embeds[]` array supports four methods: `play()`, `stop()`, `rewind()`, and `seek()`. With this information, you can write simple scripts that control how the plugin displays a movie you have embedded on a web page. Note that while some vendors may produce plugins (for Netscape) and ActiveX controls (for IE) that define the same public API, this is not always the case, and scripting embedded objects usually involves platform-specific JavaScript code.

Chapter 15. Forms and Form Elements

As we've seen in examples throughout this book, the use of HTML forms is basic to almost all JavaScript programs. This chapter explains the details of programming with forms in JavaScript. It is assumed that you are already somewhat familiar with the creation of HTML forms and with the input elements that they contain. If not, you may want to refer to a good book on HTML.^[1] The client-side reference section of this book lists the HTML syntax along with the JavaScript syntax for forms and form elements; you may find these useful for quick reference.

^[1] Such as *HTML: The Definitive Guide*, by Chuck Musciano and Bill Kennedy (O'Reilly).

If you are already familiar with server-side programming using HTML forms, you may find that things are done differently when forms are used with JavaScript. In the server-side model, a form with the input data it contains is submitted to the web server all at once. The emphasis is on processing a complete batch of input data and dynamically producing a new web page in response. With JavaScript, the programming model is quite different. In JavaScript programs, the emphasis is not on form submission and processing but instead on event handling. A form and all input elements in it have event handlers that JavaScript can use to respond to user interactions within the form. If the user clicks on a checkbox, for example, a JavaScript program can receive notification through an event handler and might respond by changing the value displayed in some other element of the form.

With server-side programs, an HTML form isn't useful unless it has a **Submit** button (or unless it has only a single text input field and allows the user to press the **Return** key as a shortcut for submission). With JavaScript, on the other hand, a **Submit** button is never necessary (unless the JavaScript program is working with a cooperating server-side program, of course). With JavaScript, a form can have any number of push buttons with event handlers that perform any number of actions when clicked. In previous chapters,

we've seen some of the possible actions that such buttons can trigger: replacing one image with another, using the `location` property to load and display a new web page, opening a new browser window, and dynamically generating a new HTML document in another window or frame. As we'll see later in this chapter, a JavaScript event handler can even trigger a form to be submitted.

As we've seen in examples throughout this book, event handlers are almost always the central element of any interesting JavaScript program. And the most commonly used event handlers (excluding the event handlers of the Link object) are those used with forms or form elements. This chapter introduces the JavaScript Form object and the various JavaScript objects that represent form elements. It concludes with an example that illustrates how you can use JavaScript to validate user input on the client before submitting it to a server-side program running on the web server.

15.1 The Form Object

The JavaScript Form object represents an HTML form. Forms are always found as elements of the `forms[]` array, which is a property of the Document object. Forms appear in this array in the order in which they appear within the document. Thus, `document.forms[0]` refers to the first form in a document. You can refer to the last form in a document with the following:

```
document.forms[document.forms.length-1]
```

The most interesting property of the Form object is the `elements[]` array, which contains JavaScript objects (of various types) that represent the various input elements of the form. Again, elements appear in this array in the same order they appear in the document. So you can refer to the third element of the second form in the document of the current window like this:

```
document.forms[1].elements[2]
```

The remaining properties of the Form object are of less importance. The `action`, `encoding`, `method`, and `target` properties correspond directly to the `action`, `encoding`, `method`, and `target` attributes of the `<form>` tag. These properties and attributes are all used to control how form data is submitted to the web server and where the results are displayed; they are therefore useful only when the form is actually submitted to a server-side program. See the client-side reference section for an explanation of the properties, or see a book on HTML or CGI programming^[2] for a thorough discussion of the attributes. What is worth noting here is that these Form properties are all read/write strings, so a JavaScript program can dynamically set their values in order to change the way the form is submitted.

^[2] Such as *CGI Programming on the World Wide Web*, by Shishir Gundavaram (O'Reilly).

In the days before JavaScript, a form was submitted with a special-purpose **Submit** button, and form elements had their values reset with a special-purpose **Reset** button. The JavaScript Form object supports two methods, `submit()` and (as of JavaScript 1.1) `reset()`, that serve the same purpose. Invoking the `submit()` method of a Form submits the form, and invoking `reset()` resets the form elements.

To accompany the `submit()` and `reset()` methods, the Form object provides the `onsubmit` event handler to detect form submission and (as of JavaScript 1.1) the `onreset` event handler to detect form resets. The `onsubmit` handler is invoked just before the form is submitted; it can cancel the submission by returning `false`. This provides an opportunity for a JavaScript program to check the user's input for errors in order to avoid submitting incomplete or invalid data over the network to a server-side program. We'll see an example of such error checking at the end of this chapter. Note that the `onsubmit` handler is triggered only by a genuine click on a **Submit** button. Calling the `submit()` method of a form does not trigger the `onsubmit` handler.

The `onreset` event handler is similar to the `onsubmit` handler. It is invoked just before the form is reset, and it can prevent the form elements from being reset by returning `false`. This allows a JavaScript program to ask for confirmation of the reset, which can be a good idea when the form is long or detailed. You might request this sort of confirmation with an event handler like the following:

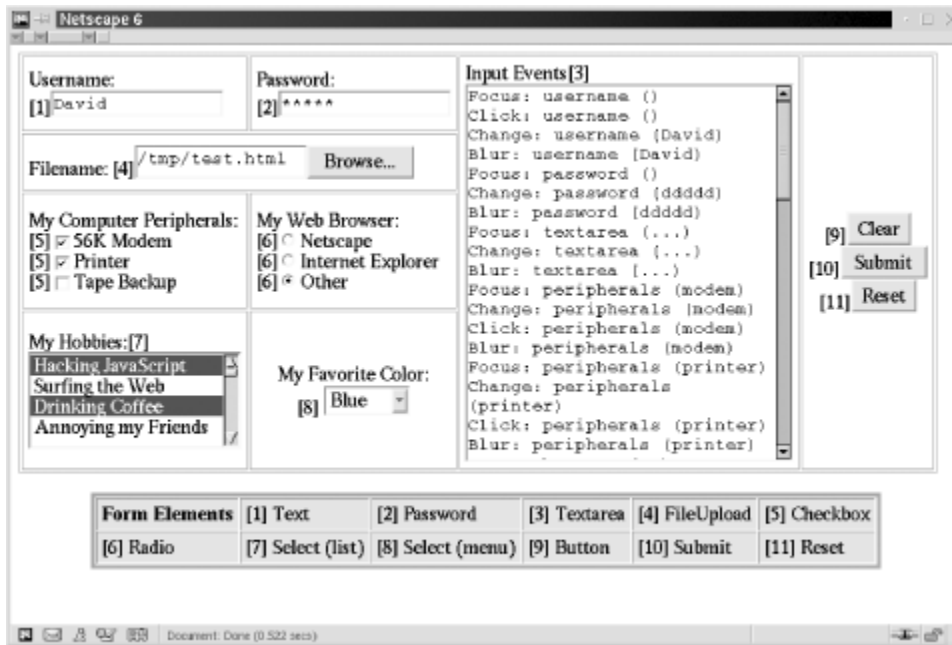
```
<form...  
    onreset="return confirm('Really erase ALL data and start over?')"  
>
```

Like the `onsubmit` handler, `onreset` is triggered only by a genuine **Reset** button. Calling the `reset()` method of a form does not trigger `onreset`.

15.2 Defining Form Elements

HTML form elements are the primitive objects with which we create graphical user interfaces for our JavaScript programs. [Figure 15-1](#) shows a complex form that contains at least one of each of the basic form elements. In case you are not already familiar with HTML form elements, the figure includes a numbered key identifying each type of element. We'll conclude this section with an example ([Example 15-1](#)) that shows the HTML and JavaScript code used to create the form pictured in [Figure 15-1](#) and to hook up event handlers to each of the form elements.

Figure 15-1. HTML form elements



[Table 15-1](#) lists the types of form elements that are available to HTML designers and JavaScript programmers. The first column of the table names the type of form element, the second column shows the HTML tags that are used to define elements of that type, and the third column lists the value of the `type` property for each type of element. As we've seen, each Form object has an `elements[]` array that contains the objects that represent the form's elements. Each of these elements has a `type` property that can be used to distinguish one type of element from another. By examining the `type` property of an unknown form element, JavaScript code can determine the type of the element and figure out what it can do with that element. (We'll see this done in [Example 15-2](#), at the end of the chapter.) Finally, the fourth column of the table provides a short description of each element and also lists the most important or most commonly used event handler for that element type.

We'll talk more about form elements later in this chapter. Complete details about the various types of elements are available in the client-side reference section, under the name listed in the first column of [Table 15-1](#). Although each type of form element has a separate reference page, note that most of the elements are created with HTML `<input>` tags and are, in fact, all Input objects. The client-side reference page named "Input" lists the features that all these elements have in common, and the type-specific pages provide specific details about working with a particular type of form element. Note that the names Button, Checkbox, and so on from the first column of the table may not correspond to any actual object in a client-side JavaScript implementation, and note also that the DOM standard does not define any interfaces with these names. Still, each type of form element has a distinct appearance and distinct behavior, and it is useful to treat them as separate types, at least for the purposes of the client-side reference section. In the DOM reference section you can find material about forms and their elements under the names

"HTMLFormElement," "HTMLInputElement," "HTMLTextAreaElement," "HTMLSelectElement," and "HTMLOptionElement."

Table 15-1. HTML form elements

Object	HTML tag	type property	Description and events
Button	<code><input type="button"></code> or <code><button type="button"></code>	"button"	A push button; <code>onclick</code> .
Checkbox	<code><input type="checkbox"></code>	"checkbox"	A toggle button without radio-button behavior; <code>onclick</code> .
FileUpload	<code><input type="file"></code>	"file"	An input field for entering the name of a file to upload to the web server; <code>onchange</code> .
Hidden	<code><input type="hidden"></code>	"hidden"	Data submitted with the form but not visible to the user; no event handlers.
Option	<code><option></code>	none	A single item within a Select object; event handlers are on the Select object, not on individual Option objects.
Password	<code><input type="password"></code>	"password"	An input field for password entry -- typed characters are not visible; <code>onchange</code> .
Radio	<code><input type="radio"></code>	"radio"	A toggle button with radio-button behavior -- only one selected at a time; <code>onclick</code> .
Reset	<code><input type="reset"></code> or <code><button type="reset"></code>	"reset"	A push button that resets a form; <code>onclick</code> .
Select	<code><select></code>	"select-one"	A list or drop-down menu from which one item may be selected; <code>onchange</code> . (See also Option object.)
Select	<code><select multiple></code>	"select-multiple"	A list from which multiple items may be selected; <code>onchange</code> . (See also Option object.)
Submit	<code><input type="submit"></code> or <code><button type="submit"></code>	"submit"	A push button that submits a form; <code>onclick</code> .

Table 15-1. HTML form elements

Object	HTML tag	type property	Description and events
Text	<code><input type="text"></code>	"text"	A single-line text entry field; onchange.
Textarea	<code><textarea></code>	"textarea"	A multiline text entry field; onchange.

Now that we've taken a look at the various types of form element and the HTML tags used to create them, [Example 15-1](#) shows the HTML code used to create the form shown in [Figure 15-1](#). Although the example consists primarily of HTML, it also contains JavaScript code used to define event handlers for each of the form elements. You'll notice that the event handlers are not defined as HTML attributes. Instead, they are JavaScript functions assigned to the properties of the objects in the form's `elements[]` array. The event handlers all call the function `report()`, which contains code that works with the various form elements. The next section of this chapter explains everything you need to know to understand what the `report()` function is doing.

Example 15-1. An HTML form containing all form elements

```
<form name="everything">                                <!-- A one-of-everything
HTML form... -->
  <table border="border" cellpadding="5">             <!-- in a big HTML table -->
    <tr>
      <td>Username:<br>[1]<input type="text" name="username"
size="15"></td>
      <td>Password:<br>[2]<input type="password" name="password"
size="15"></td>
      <td rowspan="4">Input Events[3]<br>
      <textarea name="textarea" rows="20" cols="28"></textarea></td>
      <td rowspan="4" align="center" valign="center">
      [9]<input type="button" value="Clear" name="clearbutton"><br>
      [10]<input type="submit" name="submitbutton" value="Submit"><br>
      [11]<input type="reset" name="resetbutton"
value="Reset"></td></tr>
    <tr>
      <td colspan="2">
      Filename: [4]<input type="file" name="file" size="15"></td></tr>
    <tr>
      <td>My Computer Peripherals:<br>
      [5]<input type="checkbox" name="peripherals" value="modem">56K
Modem<br>
      [5]<input type="checkbox" name="peripherals"
value="printer">Printer<br>
      [5]<input type="checkbox" name="peripherals" value="tape">Tape
Backup</td>
      <td>My Web Browser:<br>
      [6]<input type="radio" name="browser" value="nn">Netscape<br>
```

```

        [6]<input type="radio" name="browser" value="ie">Internet
Explorer<br>
        [6]<input type="radio" name="browser"
value="other">Other</td></tr>
    <tr>
        <td>My Hobbies:[7]<br>
        <select multiple="multiple" name="hobbies" size="4">
            <option value="programming">Hacking JavaScript
            <option value="surfing">Surfing the Web
            <option value="caffeine">Drinking Coffee
            <option value="annoying">Annoying my Friends
        </select></td>
        <td align="center" valign="center">My Favorite Color:<br>[8]
        <select name="color">
            <option value="red">Red            <option value="green">Green
            <option value="blue">Blue        <option value="white">White
            <option value="violet">Violet   <option value="peach">Peach
        </select></td></tr>
    </table>
</form>

<div align="center">        <!-- Another table--the key to the one
above -->
    <table border="4" bgcolor="pink" cellspacing="1" cellpadding="4">
    <tr>
        <td align="center"><b>Form Elements</b></td>
        <td>[1] Text</td> <td>[2] Password</td> <td>[3] Textarea</td>
        <td>[4] FileUpload</td> <td>[5] Checkbox</td></tr>
    <tr>
        <td>[6] Radio</td> <td>[7] Select (list)</td>
        <td>[8] Select (menu)</td> <td>[9] Button</td>
        <td>[10] Submit</td> <td>[11] Reset</td></tr>
    </table>
</div>

<script>
// This generic function appends details of an event to the big
Textarea
// element in the form above. It is called from various event handlers.
function report(element, event) {
    var elmtname = element.name;
    if ((element.type == "select-one") || (element.type == "select-
multiple")){
        value = " ";
        for(var i = 0; i < element.options.length; i++)
            if (element.options[i].selected)
                value += element.options[i].value + " ";
    }
    else if (element.type == "textarea") value = "...";
    else value = element.value;
    var msg = event + ": " + elmtname + ' (' + value + ')\n';
    var t = element.form.textarea;
    t.value = t.value + msg;
}

// This function adds a bunch of event handlers to every element in a
form.

```

```

// It doesn't bother checking to see if the element supports the event
handler,
// it just adds them all. Note that the event handlers call report( )
above.
// Note that we're defining event handlers by assigning functions to
the
// properties of JavaScript objects rather than by assigning strings to
// the attributes of HTML elements.
function addhandlers(f) {
    // Loop through all the elements in the form
    for(var i = 0; i < f.elements.length; i++) {
        var e = f.elements[i];
        e.onclick = function( ) { report(this, 'Click'); }
        e.onchange = function( ) { report(this, 'Change'); }
        e.onfocus = function( ) { report(this, 'Focus'); }
        e.onblur = function( ) { report(this, 'Blur'); }
        e.onselect = function( ) { report(this, 'Select'); }
    }

    // Define some special-case event handlers for the three buttons:
    f.clearbutton.onclick = function( ) {
        this.form.textarea.value=''; report(this, 'Click');
    }
    f.submitbutton.onclick = function ( ) {
        report(this, 'Click'); return false;
    }
    f.resetbutton.onclick = function( ) {
        this.form.reset( ); report(this, 'Click'); return false;
    }
}
// Finally, activate our form by adding all possible event handlers!
addhandlers(document.everything);
</script>

```

15.3 Scripting Form Elements

The previous section listed the form elements provided by HTML and explained how to embed these elements in your HTML documents. This section takes the next step and shows you how you can work with those elements in your JavaScript programs.

15.3.1 Naming Forms and Form Elements

Every form element has a `name` attribute that must be set in its HTML tag if the form is to be submitted to a server-side program. While form submission is not generally of interest to JavaScript programs, there is another useful reason to specify this `name` attribute, as you'll see shortly.

The `<form>` tag itself also has a `name` attribute that you can set. This attribute has nothing to do with form submission. It exists for the convenience of JavaScript programmers. If the `name` attribute is defined in a `<form>` tag, when the Form object is created for that form, it is stored as an element in the `forms[]` array of the Document object, as usual, and it is also stored in its own personal property of the Document object. The name of

this newly defined property is the value of the `name` attribute. In [Example 15-1](#), for instance, we defined a form with a tag like this:

```
<form name="everything">
```

This allowed us to refer to the Form object as:

```
document.everything
```

Often, you'll find this more convenient than the array notation:

```
document.forms[0]
```

Furthermore, using a form name makes your code position-independent: it works even if the document is rearranged so that forms appear in a different order.

``, `<applet>`, and other HTML tags also have `name` attributes that work the same as the `name` attribute of `<form>`. With forms, however, this style of naming goes a step further, because all elements contained within a form also have `name` attributes. When you give a form element a `name`, you create a new property of the Form object that refers to that element. The name of this property is the value of the attribute. Thus, you can refer to an element named "zipcode" in a form named "address" as:

```
document.address.zipcode
```

With reasonably chosen names, this syntax is much more elegant than the alternative, which relies on hardcoded (and position-dependent) array indices:

```
document.forms[1].elements[4]
```

In order for a group of Radio elements in an HTML form to exhibit mutually exclusive "radio-button" behavior, they must all be given the same `name`. In [Example 15-1](#), for instance, we define three Radio elements that all have a `name` attribute of "browser". Although it is not strictly necessary, it is also common practice to define related groups of Checkbox elements with the same `name` attribute. Sharing a `name` attribute like this works naturally for server-side programming, but it is a little awkward on the client side. The solution is straightforward, though: when more than one element in a form has the same `name` attribute, JavaScript simply places those elements into an array with the specified name. The elements of the array are in the same order as they appear in the document. So, the Radio objects in [Example 15-1](#) can be referred to as:

```
document.everything.browser[0]
```

```
document.everything.browser[1]
document.everything.browser[2]
```

15.3.2 Form Element Properties

All (or most) form elements have the following properties in common. Some elements have other special-purpose properties that are described later, when we consider the various types of form elements individually.

`type`

A read-only string that identifies the type of the form element. The third column of [Table 15-1](#) lists the value of this property for each form element.

`form`

A read-only reference to the Form object in which this element is contained.

`name`

A read-only string specified by the HTML `name` attribute.

`value`

A read/write string that specifies the "value" contained or represented by the form element. This is the string that is sent to the web server when the form is submitted, and it is only sometimes of interest to JavaScript programs. For Text and Textarea elements, this property contains the text that the user entered. For Button elements, this property specifies the text displayed within the button, which is something that you might occasionally want to change from a script. For Radio and Checkbox elements, however, the `value` property is not edited or displayed to the user in any way. It is simply a string set by the HTML `value` attribute that is passed to the web server when the form is submitted. We'll discuss the `value` property when we consider the different categories of form elements later in this chapter.

15.3.3 Form Element Event Handlers

Most form elements support most of the following event handlers:

`onclick`

Triggered when the user clicks the mouse on the element. This handler is particularly useful for Button and related form elements.

`onchange`

Triggered when the user changes the value represented by the element by entering text or selecting an option, for example. Button and related elements typically do not support this event handler because they do not have an editable value. Note that this handler is not triggered every time the user types a key in a text field, for example. It is triggered only when the user changes the value of an element and then moves the input focus to some other form element. That is, the invocation of this event handler indicates a completed change.

`onfocus`

Triggered when the form element receives the input focus.

`onblur`

Triggered when the form element loses the input focus.

[Example 15-1](#) shows how you can define these event handlers for form elements. The example is designed to report events as they occur by listing them in a large Textarea element. This makes the example a useful way to experiment with form elements and the event handlers they trigger.

An important thing to know about event handlers is that within the code of an event handler, the `this` keyword always refers to the document element that triggered the event. Since all form elements have a `form` property that refers to the containing form, the event handlers of a form element can always refer to the Form object as `this.form`. Going a step further, this means that an event handler for one form element can refer to a sibling form element named `x` as `this.form.x`.

Note that the four form element event handlers listed in this section are the ones that have particular significance for form elements. Form elements also support the various event handlers (such as `onmousedown`) that are supported by (nearly) all HTML elements. See [Chapter 19](#) for a full discussion of events and event handlers.

15.3.4 Buttons

The Button form element is one of the most commonly used, because it provides a clear visual way to allow the user to trigger some scripted action. The Button object has no default behavior of its own, and it is never useful in a form unless it has an `onclick` (or other) event handler. The `value` property of a Button element controls the text that appears within the button itself. In fourth-generation browsers, you can set this property to change the text (plain text only, not HTML) that appears in the button, which can occasionally be a useful thing to do.

Note that hyperlinks provide the same `onclick` event handler that buttons do, and any button object can be replaced with a link that does the same thing when clicked. Use a button when you want an element that looks like a graphical push button. Use a link when the action to be triggered by the `onclick` handler can be conceptualized as "following a link."

Submit and Reset elements are just like Button elements, but they have default actions (submitting and resetting a form) associated with them. Because these elements have default actions, they can be useful even without an `onclick` event handler. On the other hand, because of their default actions, they are more useful for forms that are submitted to a web server than for pure client-side JavaScript programs. If the `onclick` event handler returns `false`, the default action of these buttons is not performed. You can use the `onclick` handler of a Submit element to perform form validation, but it is more common to do this with the `onsubmit` handler of the Form object itself.

In HTML 4, you can create Button, Submit, and Reset buttons with the `<button>` tag instead of the traditional `<input>` tag. `<button>` is more flexible, because instead of simply displaying the plain text specified by the `value` attribute, it displays any HTML content (formatted text and/or images) that appears between `<button>` and `</button>`. The Button objects created by a `<button>` tag are technically different from those created by an `<input>` tag, but they have the same value for the `type` field and otherwise behave quite similarly. The main difference is that because the `<button>` tag doesn't use its `value` attribute to define the appearance of the button, you can't change that appearance by setting the `value` property. In this book, we use the terms Button, Submit, and Reset to refer to objects created with either `<input>` or `<button>`.

15.3.5 Toggle Buttons

The Checkbox and Radio elements are toggle buttons, or buttons that have two visually distinct states: they can be checked or unchecked. The user can change the state of a toggle button by clicking on it. Radio elements are designed to be used in groups of related elements, all of which have the same value for the HTML `name` attribute. Radio elements created in this way are mutually exclusive -- when you check one, the one that was previously checked becomes unchecked. Checkboxes are also often used in groups that share a `name` attribute, and when you refer to these elements by name, you must remember that the object you refer to by name is an array of same-named elements. In

[Example 15-1](#), we saw three Checkbox objects with the name "peripherals". In this example, we can refer to an array of these three Checkbox objects as:

```
document.everything.peripherals
```

To refer to an individual Checkbox element, we must index the array:

```
document.everything.peripherals[0] // First form element named  
"peripherals"
```

Radio and Checkbox elements both define a `checked` property. This read/write boolean value specifies whether the element is currently checked. The `defaultChecked` property is a read-only boolean that has the value of the HTML `checked` attribute; it specifies whether the element was checked when the page was first loaded.

Radio and Checkbox elements do not display any text themselves and are typically displayed with adjacent HTML text (or, in HTML 4, with an associated `<label>` tag.) This means that setting the `value` property of a Checkbox or Radio element does not alter the visual appearance of the element, as it does for Button elements. You can set `value`, but this changes only the string that is sent to the web server when the form is submitted.

When the user clicks on a toggle button, the Radio or Checkbox element triggers its `onclick` event handler to notify the JavaScript program of the change of state. Newer web browsers also trigger the `onchange` handler for these elements. Both event handlers convey the same essential information, but the `onclick` handler is more portable.

15.3.6 Text Fields

The Text element is probably the most commonly used element in HTML forms and JavaScript programs. It allows the user to enter a short, single-line string of text. The `value` property represents the text the user has entered. You can set this property to specify explicitly the text that should be displayed in the field. The `onchange` event handler is triggered when the user enters new text or edits existing text and then indicates that he is finished editing by moving input focus out of the text field.

The Textarea element is just like the Text element, except that it allows the user to input (and your JavaScript programs to display) multiline text. Textarea elements are created with a `<textarea>` tag using a syntax significantly different from the `<input>` tag used to create a Text element. Nevertheless, the two types of element behave quite similarly, and Textarea can be considered to inherit from `HTMLInputElement`, even though it technically does not. You can use the `value` property and `onchange` event handler of a Textarea element just as you would for a Text element.

The Password element is a modified Text element that displays asterisks as the user types into it. As the name indicates, this is useful to allow the user to enter passwords without worrying about others reading over their shoulders. Password triggers its `onchange` event handler just as Text does, but there are some restrictions (or bugs) on the use of its `value` property. Some old browsers (such as Netscape 3) implement an ineffective security measure that prevents JavaScript from reading the value the user has entered into a Password element. In other browsers (such as Netscape 4), the `value` property may be set, but setting it does not cause any change to the visual appearance of the form element. Note that the Password element protects the user's input from prying eyes, but when the form is submitted, that input is not encrypted in any way (unless it is submitted over a secure HTTPS connection), and it may be visible as it is transmitted over the network.

Finally, the FileUpload object is designed to allow the user to enter the name of a file to be uploaded to the web server. It is essentially a Text element combined with a built-in button that pops up a file-chooser dialog box. FileUpload has an `onchange` event handler, like the Text element. Unlike Text, however, the `value` property of FileUpload is read-only. This prevents malicious JavaScript programs from tricking the user into uploading a file that should not be shared.

Netscape 4 and later and Internet Explorer 4 and later define `onkeypress`, `onkeydown`, and `onkeyup` event handlers (note, however, that they are not yet part of the DOM standard). These handlers can be specified for any Document object, but they are most useful (and, in Netscape 4, only useful) when specified on Text and related form elements that actually accept keyboard input. You may return `false` from the `onkeypress` or `onkeydown` event handlers to prevent the user's keystroke from being recorded. This can be useful, for example, when you want to force the user to enter only digits. See "HTML Element" in the client-side and DOM reference sections for more details on these and other event handlers that are supported by all HTML elements.

15.3.7 Select and Option Elements

The Select element represents a set of options (represented by Option elements) from which the user can select. Browsers typically render Select elements in drop-down menus or list boxes. The Select element can operate in two very distinct ways, and the value of the `type` property depends on how it is configured. If the `<select>` tag has the `multiple` attribute, the user is allowed to select multiple options, and the `type` property of the Select object is "select-multiple". Otherwise, if the `multiple` attribute is not present, only a single item may be selected, and the `type` property is "select-one".

In some ways, a "select-multiple" element is like a set of Checkbox elements, and a "select-one" element is like a set of Radio elements. The Select element differs from the toggle-button elements in that a single Select element represents an entire set of options. These options are specified in HTML with the `<option>` tag, and they are represented in JavaScript by Option objects stored in the `options[]` array of the Select element. Because a Select element represents a set of choices, it does not have a `value` property,

as all other form elements do. Instead, as we'll discuss shortly, each Option object contained by the Select element defines a `value` property.

When the user selects or deselects an option, the Select element triggers its `onchange` event handler. For "select-one" Select elements, the read/write `selectedIndex` property specifies by number which one of the options is currently selected. For "select-multiple" elements, the single `selectedIndex` property is not sufficient to represent the complete set of selected options. In this case, to determine which options are selected you must loop through the elements of the `options[]` array and check the value of the `selected` property for each Option object.

In addition to its `selected` property, the Option element has a `text` property that specifies the string of plain text that appears in the Select element for that option. You can set this property to change the text that is displayed to the user. The `value` property is also a read/write string that specifies the text to be sent to the web server when the form is submitted. Even if you are writing a pure client-side program and your form never gets submitted, the `value` property (or its corresponding HTML `value` attribute) can be a useful place to store any data that you'll need if the user selects a particular option. Note that the Option element does not define form-related event handlers; use the `onchange` handler of the containing Select element instead.

In addition to setting the `text` property of Option objects, there are other ways you can dynamically change the options displayed in a Select element. You can truncate the array of Option elements by setting `options.length` to the desired number of options, and you can remove all Option objects by setting `options.length` to zero. Suppose we have a Select object named "country" in a form named "address". We can remove all options from the element like this:

```
document.address.country.options.length = 0; // Remove all options
```

We can remove an individual Option object from the Select element by setting its spot in the `options[]` array to `null`. This deletes the Option object, and any higher elements in the `options[]` array automatically get moved down to fill the empty spot:

```
// Remove a single Option object from the Select element
// The Option that was previously at options[11] gets moved to
options[10]...
document.address.country.options[10] = null;
```

Finally, the Option element defines an `Option()` constructor that you can use (in JavaScript 1.1 and later) to dynamically create new Option elements, and you can append new options to a Select element by assigning them to the end of the `options[]` array. For example:

```
// Create a new Option object
```

```
var zaire = new Option("Zaire", // The text property
                      "zaire", // The value property
                      false,   // The defaultSelected property
                      false);  // The selected property

// Display it in a Select element by appending it to the options array:
var countries = document.address.country; // Get the Select object
countries.options[countries.options.length] = zaire;
```

In HTML 4, you can use the `<optgroup>` tag to group related options within a Select element. The `<optgroup>` tag has a `label` attribute that specifies text to appear in the Select element. Despite its visual presence, however, an `<optgroup>` tag is not selectable by the user, and `HTMLOptGroupElement` objects never appear in the `options[]` array of the Select element.

15.3.8 Hidden Elements

As its name implies, the Hidden element has no visual representation in a form. It exists to allow arbitrary text to be transmitted to the server when a form is submitted. Server-side programs use this as a way to save state information that is passed back to them with form submission. Since they have no visual appearance, Hidden elements cannot generate events and have no event handlers. The `value` property allows you to read and write the text associated with a Hidden element, but, in general, Hidden elements are not commonly used in client-side JavaScript programming.

15.3.9 Fieldset Elements

The HTML 4 standard adds new `<fieldset>` and `<label>` tags to the set of elements that can appear within a form. In IE 5 and later, placing a `<fieldset>` in a form causes a corresponding object to be added to the form's `elements[]` array. Fieldset elements are not scriptable in interesting ways like other form elements are, and their objects do not have a `type` property like other form elements do. Therefore, the presence of Fieldset objects in the `elements[]` array seems like a mistaken design decision. This is particularly true since `<label>` tags do not cause corresponding objects to be added to the `elements[]` array. The Mozilla and Netscape 6 browsers have chosen to follow Microsoft's lead on this in order to be compatible with IE.

What this means is that if you define a form that contains fieldsets, the contents of the `elements[]` array differ in recent, HTML 4-capable browsers and in older, pre-HTML 4 browsers. In this situation, using position-based numeric indexes in the `elements[]` array is not portable, and you should define `name` attributes for all your form elements and refer to them by name.

15.4 Form Verification Example

We'll close our discussion of forms with an extended example that demonstrates several of the concepts we've been discussing. [Example 15-2](#) shows how you might use the `onsubmit` event handler of the Form object to perform input validation so that you can notify the user and prevent the form from being submitted when it contains missing or invalid data. After studying this example, you may want to turn back to [Example 1-3](#), the form-programming example we began this book with. The code of that example probably makes more sense now that you are a JavaScript expert!

[Example 15-2](#) defines a `verify()` function suitable for use as a generic form validator. It checks for required fields that are empty. In addition, it can check that a numeric value is in fact numeric and also falls within a specified numeric range. This `verify()` function relies on the `type` property of a form element to determine which kind of element it is. The function also relies on additional user-defined properties to distinguish optional fields from required fields and to specify the allowed ranges for numeric fields. Note how the function reads the `value` property of an input field and uses the `name` property of a field when reporting errors.

[Figure 15-2](#) shows an example form that uses this verification scheme, with the error message that is displayed when the user attempts to submit the form before correctly filling it in.

Figure 15-2. A form that failed validation



Example 15-2. Performing form validation

```
<script language="JavaScript1.1">
// A utility function that returns true if a string contains only
// whitespace characters
function isblank(s) {
    for(var i = 0; i < s.length; i++) {
        var c = s.charAt(i);
        if ((c != ' ') && (c != '\n') && (c != '\t')) return false;
    }
    return true;
}
```

```

}

// This is the function that performs form verification. It is invoked
// from the onsubmit event handler. The handler should return whatever
// value this function returns.
function verify(f) {
    var msg;
    var empty_fields = "";
    var errors = "";

    // Loop through the elements of the form, looking for all Text and
    // Textarea elements that don't have an "optional" property
    defined.
    // Then check for fields that are empty and make a list of them.
    Also, if
    // any of these elements have a "min" or a "max" property defined,
    verify
    // that they are numbers and are in the right range. If the element
    has a
    // "numeric" property defined, verify that it is a number, but
    don't check
    // its range. Put together error messages for fields that are
    wrong.
    for(var i = 0; i < f.length; i++) {
        var e = f.elements[i];
        if ((e.type == "text") || (e.type == "textarea")) &&
!e.optional) {
            // First check if the field is empty
            if ((e.value == null) || (e.value == "") ||
isblank(e.value)) {
                empty_fields += "\n          " + e.name;
                continue;
            }

            // Now check for fields that are supposed to be numeric
            if (e.numeric || (e.min != null) || (e.max != null)) {
                var v = parseFloat(e.value);
                if (isNaN(v) ||
                    ((e.min != null) && (v < e.min)) ||
                    ((e.max != null) && (v > e.max))) {
                    errors += "- The field " + e.name + " must be a
number";

                    if (e.min != null)
                        errors += " that is greater than " + e.min;
                    if (e.max != null && e.min != null)
                        errors += " and less than " + e.max;
                    else if (e.max != null)
                        errors += " that is less than " + e.max;
                    errors += ".\n";
                }
            }
        }
    }

    // Now, if there were any errors, display the messages, and
    // return false to prevent the form from being submitted.
    // Otherwise, return true.

```



```

        if (!empty_fields && !errors) return true;

        msg = "
_____\n\n"
        msg += "The form was not submitted because of the following
error(s).\n";
        msg += "Please correct these error(s) and re-submit.\n";
        msg += "
_____\n\n"

        if (empty_fields) {
            msg += "- The following required field(s) are empty:"
                + empty_fields + "\n";
            if (errors) msg += "\n";
        }
        msg += errors;
        alert(msg);
        return false;
    }
</script>

<!-------
---
    Here's a sample form to test our verification. Note that we
    call verify( ) from the onsubmit event handler and return
whatever
    value it returns. Also note that we use the onsubmit handler as
    an opportunity to set properties of the Form objects that verify(
)
    requires for the verification process.
-----
---->
<form onsubmit="
    this.firstname.optional = true;
    this.phonenumber.optional = true;
    this.zip.min = 0;
    this.zip.max = 99999;
    return verify(this);
">

First name: <input type="text" name="firstname">
Last name: <input type="text" name="lastname"><br>
Address:<br><textarea name="address" rows="4" cols="40"></textarea><br>
Zip Code: <input type="text" name="zip"><br>
Phone Number: <input type="text" name="phonenumber"><br>
<input type="submit">
</form>

```

Chapter 16. Scripting Cookies

The Document object contains a property named `cookie` that was not discussed in [Chapter 14](#). On the surface, this property appears to be a simple string value; however, the `cookie` property controls a very important feature of the web browser and is important enough to warrant a complete chapter of its own.

16.1 An Overview of Cookies

A *cookie* is a small amount of named data stored by the web browser and associated with a particular web page or web site.^[1] Cookies serve to give the web browser a memory, so that scripts and server-side programs can use data that was input on one page in another page, or so the browser can recall user preferences or other state variables when the user leaves a page and then returns. Cookies were originally designed for CGI programming, and at the lowest level, they are implemented as an extension to the HTTP protocol. Cookie data is automatically transmitted between the web browser and web server, so CGI scripts on the server can read and write cookie values that are stored on the client. As we'll see, JavaScript can also manipulate cookies using the `cookie` property of the Document object.

^[1] The name "cookie" does not have a lot of significance, but it is not used without precedent. In the obscure annals of computing history, the term "cookie" or "magic cookie" has been used to refer to a small chunk of data, particularly a chunk of privileged or secret data, akin to a password, that proves identity or permits access. In JavaScript, cookies are used to save state and can serve to establish a kind of identity for a web browser. Cookies in JavaScript do not use any kind of cryptography, however, and are not secure in any way.

`cookie` is a string property that allows you to read, create, modify, and delete the cookie or cookies that apply to the current web page. Although `cookie` appears at first to be a normal read/write string property, its behavior is actually more complex. When you read the value of `cookie`, you get a string that contains the names and values of all the cookies that apply to the document. You create, modify, or delete a cookie by setting the value of the `cookie` property. Later sections of this chapter explain in detail how this works. To use the `cookie` property effectively, however, you need to know more about cookies and how they work.

In addition to a name and a value, each cookie has four optional attributes that control its lifetime, visibility, and security. The first attribute is `expires`, which specifies the cookie's lifetime. Cookies are transient by default -- the values they store last for the duration of the web-browser session but are lost when the user exits the browser. If you want a cookie to last beyond a single browsing session, you use its `expires` attribute to specify an expiration date -- this attribute causes the browser to save the cookie in a local file so that it can read it back in the next time the user visits the web page. Once the expiration date has passed, the cookie is automatically deleted from the cookie file.

The second attribute of a cookie is `path`, which specifies the web pages with which a cookie is associated. By default, a cookie is associated with, and accessible to, the web page that created it and any other web pages in the same directory or any subdirectories

of that directory. If the web page *http://www.acme.com/catalog/index.html* creates a cookie, for example, that cookie is also visible to *http://www.acme.com/catalog/order.html* and *http://www.acme.com/catalog/widgets/index.html*, but it is not visible to *http://www.acme.com/about.html*.

This default visibility behavior is often exactly what you want. Sometimes, though, you'll want to use cookie values throughout a multipage web site, regardless of which page creates the cookie. For instance, if the user enters his mailing address in a form on one page, you may want to save that address to use as the default the next time he returns to the page and also as the default in an entirely unrelated form on another page where he is asked to enter a billing address. To allow this usage, you specify a `path` for the cookie. Then, any web page from the same web server that contains that path in its URL can share the cookies. For example, if a cookie set by *http://www.acme.com/catalog/widgets/index.html* has its path set to `"/catalog"`, that cookie is also visible to *http://www.acme.com/catalog/order.html*. Or, if the path is set to `"/"`, the cookie is visible to any page on the *www.acme.com* web server.

By default, cookies are accessible only to pages on the same web server from which they were set. Large web sites may want cookies to be shared across multiple web servers, however. For example, the server at *order.acme.com* may need to read cookie values set from *catalog.acme.com*. This is where the third cookie attribute, `domain`, comes in. If a cookie created by a page on *catalog.acme.com* sets its `path` attribute to `"/"` and its `domain` attribute to `".acme.com"`, that cookie is available to all web pages on *catalog.acme.com*, *orders.acme.com*, and any other server in the *acme.com* domain. If the `domain` attribute is not set for a cookie, the default is the hostname of the web server that serves the page. Note that you cannot set the domain of a cookie to a domain other than the domain of your server.

The fourth and final cookie attribute is a boolean attribute named `secure` that specifies how cookie values are transmitted over the network. By default, cookies are insecure, which means that they are transmitted over a normal, insecure HTTP connection. If a cookie is marked secure, however, it is transmitted only when the browser and server are connected via HTTPS or another secure protocol.

Note that the `expires`, `path`, `domain`, and `secure` attributes of a cookie are not JavaScript object properties. We'll see later in the chapter how you set these cookie attributes.

If you are interested in the complete technical details of how cookies work, see http://www.netscape.com/newsref/std/cookie_spec.html. This document is the original specification for HTTP cookies; it contains low-level details that are more suitable to CGI programming than to JavaScript programming. The following sections discuss how you can set and query cookie values in JavaScript and how you can specify the `expires`, `path`, `domain`, and `secure` attributes of a cookie.

16.2 Storing Cookies

To associate a transient cookie value with the current document, simply set the `cookie` property to a string of the form:

```
name=value
```

For example:

```
document.cookie = "version=" + escape(document.lastModified);
```

The next time you read the `cookie` property, the name/value pair you stored is included in the list of cookies for the document. Cookie values may not include semicolons, commas, or whitespace. For this reason, you may want to use the JavaScript `escape()` function to encode the value before storing it in the cookie. If you do this, you'll have to use the corresponding `unescape()` function when you read the cookie value.

A cookie written as described above lasts for the current web-browsing session but is lost when the user exits the browser. To create a cookie that can last across browser sessions, include an expiration date by setting the `expires` attribute. You can do this by setting the `cookie` property to a string of the form:

```
name=value; expires=date
```

When setting an expiration date like this, `date` should be a date specification in the format written by `Date.toGMTString()`. For example, to create a cookie that persists for a year, you can use code like this:

```
var nextyear = new Date( );  
nextyear.setFullYear(nextyear.getFullYear( ) + 1);  
document.cookie = "version=" + document.lastModified +  
    "; expires=" + nextyear.toGMTString( );
```

Similarly, you can set the `path`, `domain`, and `secure` attributes of a cookie by appending strings of the following format to the cookie value before that value is written to the `cookie` property:

```
; path=path  
; domain=domain  
; secure
```

To change the value of a cookie, set its value again, using the same name and the new value. Use whatever values are appropriate for `expires`, `path`, and the other attributes. To delete a cookie, set it again using the same name, an arbitrary value, and an expiration date that has already passed. Note that the browser is not required to delete expired

cookies immediately, so a cookie may remain in the browser's cookie file past its expiration date.

16.2.1 Cookie Limitations

Cookies are intended for infrequent storage of small amounts of data. They are not intended as a general-purpose communication or data-transfer mechanism, so you should use them in moderation. Web browsers are not required to retain more than 300 cookies total, 20 cookies per web server (for the entire server, not just for your page or site on the server), or 4 kilobytes of data per cookie (both name and value count toward this 4-kilobyte limit). The most restrictive of these is the 20 cookies per server limit. In order to avoid reaching that limit, you may want to avoid using a separate cookie for each state variable you want to save. Instead, you can encode several related state variables into a single named cookie. [Example 16-1](#), later in this chapter, shows one way that this can be done.

16.3 Reading Cookies

When you use the `cookie` property in a JavaScript expression, the value it returns is a string that contains all the cookies that apply to the current document.^[2] The string is a list of `name=value` pairs separated by semicolons, where `name` is the name of a cookie and `value` is its string value. This value does not include any of the attributes that may have been set for the cookie. To determine the value of a particular named cookie, you can use the `String.indexOf()` and `String.substring()` methods, or you can use `String.split()` to break the string into individual cookies.

^[2] In Internet Explorer 3, the `cookie` property works only for Document objects that were retrieved using the HTTP protocol. Documents retrieved from the local filesystem or via other protocols, such as FTP, cannot have cookies associated with them.

Once you have extracted the value of a cookie from the `cookie` property, you must interpret that value based on whatever format or encoding was used by the cookie's creator. For example, the cookie might store multiple pieces of information in colon-separated fields. In this case, you would have to use appropriate string methods to extract the various fields of information. Don't forget to use the `unescape()` function on the cookie value if it was encoded using the `escape()` function.

The following code shows how you might read the `cookie` property, extract a single cookie from it, and use the value of that cookie:

```
// Read the cookie property. This returns all cookies for this
document.
var allcookies = document.cookie;
// Look for the start of the cookie named "version"
var pos = allcookies.indexOf("version=");

// If we find a cookie by that name, extract and use its value
if (pos != -1) {
    var start = pos + 8;                // Start of cookie value
```

```

var end = allcookies.indexOf(";", start); // End of cookie value
if (end == -1) end = allcookies.length;
var value = allcookies.substring(start, end); // Extract the value
value = unescape(value); // Decode it

// Now that we have the cookie value, we can use it.
// In this case, the cookie was previously set to the modification
// date of the document, so we can use it to see if the document
has
// changed since the user last visited.
if (value != document.lastModified)
    alert("This document has changed since you were last here");
}

```

Note that the string returned when you read the value of the `cookie` property does not contain any information about the various cookie attributes. The `cookie` property allows you to set those attributes, but it does not allow you to read them.

16.4 Cookie Example

[Example 16-1](#) brings together all the aspects of cookies we have discussed so far. First, the example defines a `Cookie` class. When you create a `Cookie` object, you specify a `Document` object, a name for the cookie, and, optionally, an expiration time, a path, a domain, and a boolean value that indicates whether the cookie should be secure. After creating a `Cookie` object, you can set arbitrary string properties on this object; the values of these properties are the values stored in the cookie.

The `Cookie` class defines three methods. The `store()` method loops through all of the user-defined properties of the `Cookie` object and concatenates their names and values into a single string that serves as the value of the cookie. The `load()` method of a `Cookie` object reads the `cookie` property of the `Document` object to obtain the values of all the cookies for the document. It searches this string to find the value of the named cookie and then parses this value into individual names and values, which it stores as properties of the `Cookie` object. Finally, the `remove()` method of the `Cookie` object deletes the specified cookie from the document.

After defining the `Cookie` class, [Example 16-1](#) demonstrates a useful and elegant way to use cookies. The code is somewhat complicated but is worth studying carefully. You may want to start with the test program at the end of the example; it shows a typical usage of the `Cookie` class.

Example 16-1. A utility class for working with cookies

```

<script language="JavaScript1.1">
// The constructor function: creates a Cookie object for the specified
// document, with a specified name and optional attributes.
// Arguments:
//   document: The Document object for which the cookie is stored.
// Required.

```

```

// name:      A string that specifies a name for the cookie. Required.
// hours:     An optional number that specifies the number of hours
from now
//           after which the cookie should expire.
// path:      An optional string that specifies the cookie path
attribute.
// domain:    An optional string that specifies the cookie domain
attribute.
// secure:    An optional boolean value that, if true, requests a
secure cookie.
//
function Cookie(document, name, hours, path, domain, secure)
{
    // All the predefined properties of this object begin with '$'
    // to distinguish them from other properties, which are the values
to
    // be stored in the cookie
    this.$document = document;
    this.$name = name;
    if (hours)
        this.$expiration = new Date((new Date()).getTime( ) +
hours*3600000);
    else this.$expiration = null;
    if (path) this.$path = path; else this.$path = null;
    if (domain) this.$domain = domain; else this.$domain = null;
    if (secure) this.$secure = true; else this.$secure = false;
}

// This function is the store( ) method of the Cookie object
Cookie.prototype.store = function ( ) {
    // First, loop through the properties of the Cookie object and
    // put together the value of the cookie. Since cookies use the
    // equals sign and semicolons as separators, we'll use colons
    // and ampersands for the individual state variables we store
    // within a single cookie value. Note that we escape the value
    // of each state variable, in case it contains punctuation or other
    // illegal characters.
    var cookieval = "";
    for(var prop in this) {
        // Ignore properties with names that begin with '$' and also
methods
        if ((prop.charAt(0) == '$') || ((typeof this[prop]) ==
'function'))
            continue;
        if (cookieval != "") cookieval += '&';
        cookieval += prop + ':' + escape(this[prop]);
    }

    // Now that we have the value of the cookie, put together the
    // complete cookie string, which includes the name and the various
    // attributes specified when the Cookie object was created
    var cookie = this.$name + '=' + cookieval;
    if (this.$expiration)
        cookie += '; expires=' + this.$expiration.toGMTString( );
    if (this.$path) cookie += '; path=' + this.$path;
    if (this.$domain) cookie += '; domain=' + this.$domain;
    if (this.$secure) cookie += '; secure';
}

```

```

    // Now store the cookie by setting the magic Document.cookie
property
    this.$document.cookie = cookie;
}

// This function is the load( ) method of the Cookie object
Cookie.prototype.load = function( ) {
    // First, get a list of all cookies that pertain to this document
    // We do this by reading the magic Document.cookie property
    var allcookies = this.$document.cookie;
    if (allcookies == "") return false;

    // Now extract just the named cookie from that list
    var start = allcookies.indexOf(this.$name + '=');
    if (start == -1) return false; // Cookie not defined for this
page
    start += this.$name.length + 1; // Skip name and equals sign
    var end = allcookies.indexOf(';', start);
    if (end == -1) end = allcookies.length;
    var cookieval = allcookies.substring(start, end);

    // Now that we've extracted the value of the named cookie, we
    // must break that value down into individual state variable
    // names and values. The name/value pairs are separated from each
    // other by ampersands, and the individual names and values are
    // separated from each other by colons. We use the split( ) method
    // to parse everything.
    var a = cookieval.split('&'); // Break it into an array of
name/value pairs
    for(var i=0; i < a.length; i++) // Break each pair into an array
        a[i] = a[i].split(':');

    // Now that we've parsed the cookie value, set all the names and
values
    // of the state variables in this Cookie object. Note that we
unescape( )
    // the property value, because we called escape( ) when we stored
it.
    for(var i = 0; i < a.length; i++) {
        this[a[i][0]] = unescape(a[i][1]);
    }

    // We're done, so return the success code
    return true;
}

// This function is the remove( ) method of the Cookie object
Cookie.prototype.remove = function( ) {
    var cookie;
    cookie = this.$name + '=';
    if (this.$path) cookie += '; path=' + this.$path;
    if (this.$domain) cookie += '; domain=' + this.$domain;
    cookie += '; expires=Fri, 02-Jan-1970 00:00:00 GMT';

    this.$document.cookie = cookie;
}

```



```

//=====
// The previous code is the definition of the Cookie class.
// The following code is a sample use of that class.
//=====

// Create the cookie we'll use to save state for this web page.
// Since we're using the default path, this cookie will be accessible
// to all web pages in the same directory as this file or "below" it.
// Therefore, it should have a name that is unique among those pages.
// Note that we set the expiration to 10 days in the future.
var visitordata = new Cookie(document, "name_color_count_state", 240);

// First, try to read data stored in the cookie. If the cookie is not
// defined, or if it doesn't contain the data we need, then query the
// user for that data.
if (!visitordata.load( ) || !visitordata.name || !visitordata.color) {
    visitordata.name = prompt("What is your name:", "");
    visitordata.color = prompt("What is your favorite color:", "");
}

// Keep track of how many times this user has visited the page:
if (visitordata.visits == null) visitordata.visits = 0;
visitordata.visits++;

// Store the cookie values, even if they were already stored, so that
the
// expiration date will be reset to 10 days from this most recent
visit.
// Also, store them again to save the updated visits state variable.
visitordata.store( );

// Now we can use the state variables we read:
document.write('<font size="7" color="' + visitordata.color + '>' +
    'Welcome, ' + visitordata.name + '!' +
    '</font>' +
    '<p>You have visited ' + visitordata.visits + '
times.');
```

```

</script>

<form>
<input type="button" value="Forget My Name"
onclick="visitordata.remove( );">
</form>

```

Chapter 17. The Document Object Model

A *document object model* (DOM) is an application programming interface (API) for representing a document (such as an HTML document) and accessing and manipulating the various elements (such as HTML tags and strings of text) that make up that document. JavaScript-enabled web browsers have always defined a document object model; a web-browser DOM may specify, for example, that the forms in an HTML document are accessible through the `forms[]` array of the Document object.

In this chapter, we'll discuss the W3C DOM, a standard document object model defined by the World Wide Web Consortium and implemented (at least partially) by Netscape 6 and Internet Explorer 5 and 6. This DOM standard^[1] is a full-featured superset of the traditional web-browser DOM. It represents HTML (and XML) documents in a tree structure and defines properties and methods for traversing the tree and examining and modifying its nodes. Other portions of the standard specify techniques for defining event handlers for the nodes of a document, working with the style sheets of a document, and manipulating contiguous ranges of a document.

^[1] Technically, the W3C issues "recommendations." These recommendations serve the same purpose and carry the same weight as international standards do, however, and are called "standards" in this book.

This chapter begins with an overview of the DOM standard and then describes the core portions of the standard for working with HTML documents. The discussion of the core standard is followed by short sections that explain the DOM-like features of Internet Explorer 4 and Netscape 4. The chapter ends with an overview of two optional parts of the DOM standard that are closely related to the core. Later chapters cover advanced DOM features for working with style sheets and events.

17.1 An Overview of the DOM

The DOM API is not particularly complicated, but before we can begin our discussion of programming with the DOM, there are a number of things you should understand about the DOM architecture.

17.1.1 Representing Documents as Trees

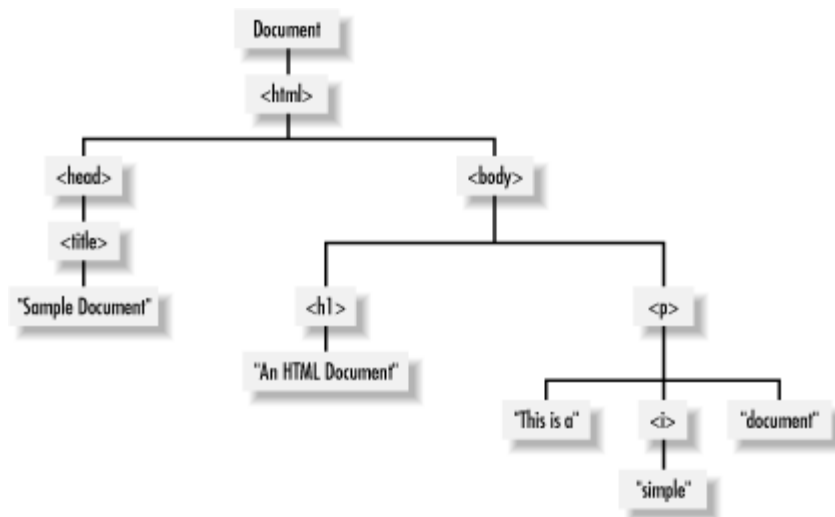
HTML documents have a hierarchical structure that is represented in the DOM as a tree structure. The nodes of the tree represent the various types of content in a document. The tree representation of an HTML document primarily contains nodes representing elements or tags such as `<body>` and `<p>` and nodes representing strings of text. An HTML document may also contain nodes representing HTML comments.^[2] Consider the following simple HTML document:

^[2] The DOM can also be used to represent XML documents, which have a more complex syntax than HTML documents, and the tree representation of such a document may contain nodes that represent XML entity references, processing instructions, CDATA sections, and so on. Most client-side JavaScript programmers do not need to use the DOM with XML documents, and although the XML-specific features of the DOM are covered in the DOM reference section, they are not emphasized in this chapter.

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

The DOM representation of this document is the tree pictured in [Figure 17-1](#).

Figure 17-1. The tree representation of an HTML document



If you are not already familiar with tree structures in computer programming, it is helpful to know that they borrow terminology from family trees. The node directly above a node is the *parent* of that node. The nodes one level directly below another node are the *children* of that node. Nodes at the same level, and with the same parent, are *siblings*. The set of nodes any number of levels below another node are the *descendants* of that node. And the parent, grandparent, and all other nodes above a node are the *ancestors* of that node.

17.1.2 Nodes

The DOM tree structure illustrated in [Figure 17-1](#) is represented as a tree of various types of Node objects. The Node interface^[3] defines properties and methods for traversing and manipulating the tree. The `childNodes` property of a Node object returns a list of children of the node, and the `firstChild`, `lastChild`, `nextSibling`,

`previousSibling`, and `parentNode` properties provide a way to traverse the tree of nodes. Methods such as `appendChild()`, `removeChild()`, `replaceChild()`, and `insertBefore()` enable you to add and remove nodes from the document tree. We'll see examples of the use of these properties and methods later in this chapter.

^[3] The DOM standard defines interfaces, not classes. If you are not familiar with the term *interface* in object-oriented programming, you can think of it as an abstract kind of class. We'll describe the difference in more detail later in this DOM overview.

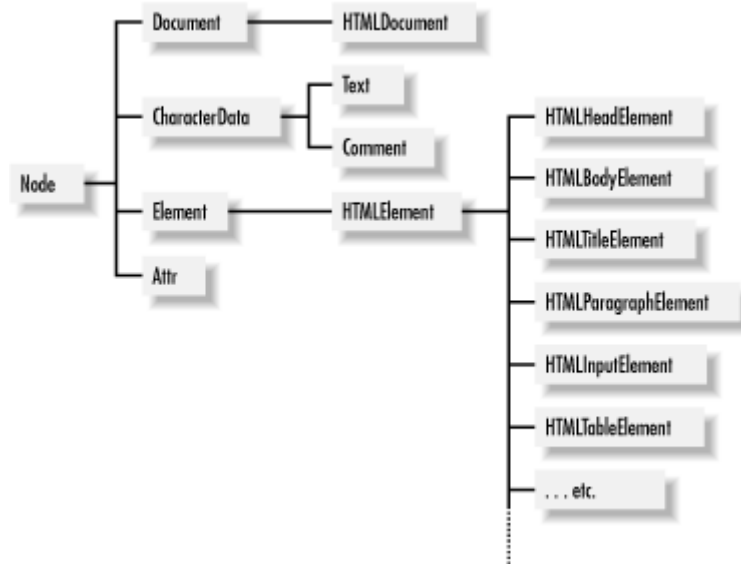
17.1.2.1 Types of nodes

Different types of nodes in the document tree are represented by specific subinterfaces of `Node`. Every `Node` object has a `nodeType` property that specifies what kind of node it is. If the `nodeType` property of a node equals the constant `Node.ELEMENT_NODE`, for example, you know the `Node` object is also an `Element` object and you can use all the methods and properties defined by the `Element` interface with it. [Table 17-1](#) lists the node types commonly encountered in HTML documents and the `nodeType` value for each one.

Table 17-1. Common node types		
Interface	nodeType constant	nodeType value
Element	<code>Node.ELEMENT_NODE</code>	1
Text	<code>Node.TEXT_NODE</code>	3
Document	<code>Node.DOCUMENT_NODE</code>	9
Comment	<code>Node.COMMENT_NODE</code>	8
DocumentFragment	<code>Node.DOCUMENT_FRAGMENT_NODE</code>	11
Attr	<code>Node.ATTRIBUTE_NODE</code>	2

The `Node` at the root of the DOM tree is a `Document` object. The `documentElement` property of this object refers to an `Element` object that represents the root element of the document. For HTML documents, this is the `<html>` tag that is either explicit or implicit in the document. (The `Document` node may have other children, such as `Comment` nodes, in addition to the root element.) The bulk of a DOM tree consists of `Element` objects, which represent tags such as `<html>` and `<i>`, and `Text` objects, which represent strings of text. If the document parser preserves comments, those comments are represented in the DOM tree by `Comment` objects. [Figure 17-2](#) shows a partial class hierarchy for these and other core DOM interfaces.

Figure 17-2. A partial class hierarchy of the core DOM API



17.1.2.2 Attributes

The attributes of an element (such as the `src` and `width` attributes of an `` tag) may be queried, set, and deleted using the `getAttribute()`, `setAttribute()`, and `removeAttribute()` methods of the `Element` interface.

Another, more awkward way to work with attributes is with the `getAttributeNode()` method, which returns an `Attr` object representing an attribute and its value. (One reason to use this more awkward technique is that the `Attr` interface defines a `specified` property that allows you to determine whether the attribute is literally specified in the document, or whether its value is a default value.) The `Attr` interface appears in [Figure 17-2](#), and it is a type of node. Note, however, that `Attr` objects do not appear in the `childNodes[]` array of an element and are not directly part of the document tree in the way that `Element` and `Text` nodes are. The DOM specification allows `Attr` nodes to be accessed through the `attributes[]` array of the `Node` interface, but Microsoft's Internet Explorer defines a different and incompatible `attributes[]` array that makes it impossible to use this feature portably.

17.1.3 The DOM HTML API

The DOM standard was designed for use with both XML and HTML documents. The core DOM API -- the `Node`, `Element`, `Document`, and other interfaces -- are relatively generic and apply to both types of documents. The DOM standard also includes interfaces that are specific to HTML documents. As you can see from [Figure 17-2](#), `HTMLDocument` is an HTML-specific subinterface of `Document`, and `HTMLElement` is an HTML-specific subinterface of `Element`. Furthermore, the DOM defines tag-specific interfaces for many HTML elements. These tag-specific interfaces, such as

HTMLBodyElement and HTMLTitleElement, typically define a set of properties that mirror the HTML tag's attributes.

The HTMLDocument interface defines various document properties and methods that were supported by browsers prior to W3C standardization. These include the `location` property, `forms[]` array, and `write()` method, which are described in [Chapter 13](#), [Chapter 14](#), and [Chapter 15](#).

The HTMLInputElement interface defines `id`, `style`, `title`, `lang`, `dir`, and `className` properties. These properties allow convenient access to the values of the `id`, `style`, `title`, `lang`, `dir`, and `class` attributes, which are allowed on all HTML tags. A number of HTML tags, listed in [Table 17-2](#), accept no attributes other than these six, and so are fully represented by the HTMLInputElement interface.

<code><abbr></code>	<code><acronym></code>	<code><address></code>	<code></code>	<code><bdo></code>
<code><big></code>	<code><center></code>	<code><cite></code>	<code><code></code>	<code><dd></code>
<code><dfn></code>	<code><dt></code>	<code></code>	<code><i></code>	<code><kbd></code>
<code><noframes></code>	<code><noscript></code>	<code><s></code>	<code><samp></code>	<code><small></code>
<code></code>	<code><strike></code>	<code></code>	<code><sub></code>	<code><sup></code>
<code><tt></code>	<code><u></code>	<code><var></code>		

All other HTML tags have corresponding interfaces defined by the HTML portion of the DOM specification. For many HTML tags, these interfaces do nothing more than provide a set of properties that mirror their HTML attributes. For example, the `` tag has a corresponding HTMLListElement interface, and the `<body>` tag has a corresponding HTMLBodyElement interface. Because these interfaces simply define properties that are standardized by the HTML standard, they are not documented in detail in this book. You can safely assume that the HTMLInputElement object that represents a particular HTML tag has properties for each of the standard attributes for that tag (but see the naming conventions described in the next section). Note that the DOM standard defines properties for HTML attributes as a "convenience" to script writers. The general (and possibly preferred) way to query and set attribute values is with the `getAttribute()` and `setAttribute()` methods of the Element object.

Some of the interfaces defined in the HTML DOM define additional properties or methods, other than those that mirror HTML attribute values. For example, the `HTMLInputElement` interface defines `focus()` and `blur()` methods, and the `HTMLFormElement` interface defines `submit()` and `reset()` methods and a `length` property. Methods and properties like these typically predate DOM standardization and have been made part of the DOM standard for backward compatibility with existing practice. Interfaces like these are documented in the DOM reference section. You can usually also find information about the "existing practice" portions of these interfaces in the client-side reference section, although this information is typically referenced under a name that also predates DOM standardization; for example, you can find information about `HTMLFormElement` and `HTMLInputElement` in the client-side reference section under "Form" and "Input."

17.1.3.1 HTML naming conventions

When working with the HTML-specific portions of the DOM standard, you should be aware of some simple naming conventions. Properties of the HTML-specific interfaces begin with lowercase letters. If the property name consists of multiple words, the first letters of the second and subsequent words are capitalized. Thus, the `maxLength` attribute of the `<input>` tag translates into the `maxLength` property of `HTMLInputElement`.

When an HTML attribute name conflicts with a JavaScript keyword, it is prefixed with the string "html" to avoid the conflict. Thus, the `for` attribute of the `<label>` tag translates to the `htmlFor` property of the `HTMLLabelElement`. An exception to this rule is the `class` attribute (which can be specified for any HTML element); it translates to the `className` property of `HTMLElement`.^[4]

^[4] The name `className` is misleading, because in addition to specifying a single class name, this property (and the HTML attribute it represents) can also specify a space-separated list of class names.

17.1.4 DOM Levels and Features

There are two versions, or "levels," of the DOM standard. DOM Level 1 was standardized in October, 1998. It defines the core DOM interfaces, such as `Node`, `Element`, `Attr`, and `Document`, and also defines various HTML-specific interfaces. DOM Level 2 was standardized in November, 2000.^[5] In addition to some updates to the core interfaces, this new version of the DOM is greatly expanded to define standard APIs for working with document events and CSS style sheets and to provide additional tools for working with ranges of documents. As of this writing, the DOM Working Group at the W3C is working to standardize DOM Level 3. You may also sometimes see a reference to DOM Level 0. This term does not refer to any formal standard but is used to refer informally to the common features of the HTML document object models implemented by Netscape and Internet Explorer prior to W3C standardization.

^[5] Except for the HTML-specific portions of the standard, which are still at the "working draft" stage as of November 2001. Fortunately, the current working draft is presumed stable and includes only minor changes (documented in this book) from the HTML-specific portions of the Level 1 standard.

As of Level 2, the DOM standard has been "modularized." The core module, which defines the basic tree structure of a document with the Document, Node, Element, and Text interfaces (among others), is the only required module. All other modules are optional and may or may not be supported, depending on the needs of the implementation. The DOM implementation of a web browser would obviously support the HTML module, since web documents are written in HTML. Browsers that support CSS style sheets typically support the StyleSheets and CSS modules, because (as we'll see in [Chapter 18](#)) CSS styles play a crucial role in Dynamic HTML programming. Similarly, since almost all interesting client-side JavaScript programming requires event-handling capabilities, you would expect web browsers to support the Events module of the DOM specification. Unfortunately, the Events module was only recently defined by the DOM Level 2 specification and is not yet widely supported at the time of this writing. We'll see a complete list of DOM Level 2 modules in the next section.

17.1.5 DOM Conformance

At the time of this writing, no browser is completely conformant to the DOM standard. Recent releases of Mozilla come closest, and complete DOM Level 2 conformance is a goal of the Mozilla project. Netscape 6.1 does a good job of conforming to the most important Level 2 modules, and Netscape 6.0 does an adequate job but has gaps in its coverage. Internet Explorer 6 is mostly compliant (with at least one annoying exception) with the Level 1 DOM, but does not support many of the Level 2 modules -- most notably the Events module, which is the topic of [Chapter 19](#). Internet Explorer 5 and 5.5 have substantial gaps in their conformance but support key DOM Level 1 methods well enough to run most of the examples in this chapter. The Macintosh version of IE 5 has considerably better support for the DOM than the Windows version of IE 5.

In addition to Mozilla, Netscape, and Internet Explorer, several other browsers offer at least partial support for the DOM. The number of available browsers has become too large, and the rate of change in the area of standards support has grown too fast, for this book to even attempt to provide definitive statements about which browsers support which particular DOM features. Therefore, you'll have to rely on other information sources to determine exactly how conformant the DOM implementation in any particular web browser is.

One source for conformance information is the implementation itself. In a conformant implementation, the `implementation` property of the Document object refers to a DOMImplementation object that defines a method named `hasFeature()`. You can use this method (if it exists) to ask an implementation whether it supports a specific feature (or module) of the DOM standard. For example, to determine whether the DOM implementation in a web browser supports the basic DOM Level 1 interfaces for working with HTML documents, you could use the following code:

```
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("html", "1.0")) {
    // The browser claims to support Level 1 Core and HTML interfaces
```


}

The `hasFeature()` method takes two arguments: the first is the name of the feature to check, and the second is a version number, expressed as a string. It returns `true` if the specified version of the specified feature is supported. [Table 17-3](#) lists the feature name/version number pairs that are defined by the DOM Level 1 and Level 2 standards. Note that the feature names are case-insensitive: you can capitalize them any way you choose. The fourth column of the table specifies what other features are required for support of a feature and are therefore implied by a return value of `true`. For example, if `hasFeature()` indicates that the `MouseEvents` module is supported, this implies that `UIEvents` is also supported, which in turn implies that the `Events`, `Views`, and `Core` modules are supported.

Table 17-3. Features that can be tested with `hasFeature()`

Feature name	Version	Description	Implies
HTML	1.0	Level 1 Core and HTML interfaces	
XML	1.0	Level 1 Core and XML interfaces	
Core	2.0	Level 2 Core interfaces	
HTML	2.0	Level 2 HTML interfaces	Core
XML	2.0	Level 2 XML-specific interfaces	Core
Views	2.0	AbstractView interface	Core
StyleSheets	2.0	Generic style-sheet traversal	Core
CSS	2.0	CSS styles	Core, Views
CSS2	2.0	CSS2Properties interface	CSS
Events	2.0	Event-handling infrastructure	Core
UIEvents	2.0	User-interface events (plus Events and Views)	Events, Views
MouseEvents	2.0	Mouse events	UIEvents
HTMLEvents	2.0	HTML events	Events
MutationEvents	2.0	Document mutation events	Events
Range	2.0	Document range interfaces	Core
Traversal	2.0	Document traversal interfaces	Core

In Internet Explorer 6 (on Windows), `hasFeature()` returns `true` only for the feature HTML and Version 1.0. It does not report compliance to any of the other features listed in [Table 17-3](#) (although, as we'll see in [Chapter 18](#), it supports the most common uses of the CSS2 module.) In Netscape 6.1, `hasFeature()` returns `true` for most feature names and version numbers, with the notable exceptions of the Traversal and MutationEvents features. It returns `false` for the Core and CSS2 features with Version 2.0, indicating incomplete support (even though support for these features is quite good).

This book documents the interfaces that make up all of the DOM modules listed in [Table 17-3](#). The Core, HTML, Traversal, and Range modules are covered in this chapter. The StyleSheets, CSS, and CSS2 modules are covered in [Chapter 18](#), and the various Event modules (except MutationEvents) are covered in [Chapter 19](#). The DOM reference section includes complete coverage of all modules.

The `hasFeature()` method is not always perfectly reliable. As previously noted, IE 6 reports Level 1 compliance to HTML features even though there are some problems with its compliance. On the other hand, Netscape 6.1 reports noncompliance to the Level 2 Core feature even though it is mostly compliant. In both cases, you need more detailed information about exactly what is and is not compliant. This is exactly the type of information that is too voluminous and volatile to include in a printed book.

If you are an active web developer, you undoubtedly already know or will discover many browser-specific support details on your own. There are also resources on the Web that can help you. Most importantly, the W3C (in collaboration with the U.S. National Institute of Standards and Technology) is working on developing an open source test suite for DOM implementations. At the time of this writing, the test suite effort is just getting off the ground, but it ought to prove to be an invaluable resource for fine-grained compliance testing of DOM implementations. See <http://www.w3c.org/DOM/Test/> for details.

The Mozilla organization has a set of test suites for a variety of standards, including DOM Level 1 (available at http://www.mozilla.org/quality/browser_sc.html). Netscape has published a test suite that includes some DOM Level 2 tests (available at <http://developer.netscape.com/evangelism/tools/testsuites/>). Netscape has also published a partisan (and dated) comparison of DOM compliance of an early Mozilla release versus IE 5.5 (available at <http://home.netscape.com/browsers/future/standards.html>). Finally, you can also find compatibility and compliance information at independent sites on the Web. One notable site is published by Peter-Paul Koch. You can find a link to his DOM Compatibility Table from his main JavaScript page (<http://www.xs4all.nl/~ppk/js/>).

17.1.5.1 DOM conformance in Internet Explorer

Because IE is the most widely used web browser, a few special notes about its compliance to the DOM specifications are appropriate here. IE 5 and later versions support the Level 1 Core and HTML features well enough to run the examples in this

chapter, and they support the key Level 2 CSS features well enough to run most of the examples in [Chapter 18](#). Unfortunately, IE 5, 5.5, and 6 do not support the DOM Level 2 Events module, even though Microsoft participated in the definition of this module and had ample time to implement it for IE 6. As we'll see in [Chapter 19](#), event handling is crucial for client-side event handling, and IE's lack of support for the standard event model impedes the development of advanced client-side web applications.

Although IE 6 claims (through its `hasFeature()` method) to support the Core and HTML interfaces of the DOM Level 1 standard, this support is actually incomplete. The most egregious problem, and the one you are most likely to encounter, is a minor but annoying one: IE does not support the node-type constants defined by the Node interface. Recall that each node in a document has a `nodeType` property that specifies what type of node it is. The DOM specification also says that the Node interface defines constants that represent each of the defined node types. For example, the constant `Node.ELEMENT_NODE` represents an Element node. In IE (at least as high as version 6), these constants simply do not exist.

The examples in this chapter have been modified to work around this problem by using integer literals instead of the corresponding symbolic constants. For example, you'll see code like this:

```
if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // Check if n is an Element
```

It is good programming style to use constants instead of hardcoded integer literals in your code, and if you'd like to do this portably, you can include the following code in your programs to define these constants if they are missing:

```
if (!window.Node) {
    var Node = {
        ELEMENT_NODE: 1, // If there is no Node object, define one
                        // with the following properties and
values.
        ATTRIBUTE_NODE: 2, // Note that these are HTML node types
only.
        TEXT_NODE: 3, // For XML-specific nodes, you need to add
COMMENT_NODE: 8, // other constants here.
        DOCUMENT_NODE: 9,
        DOCUMENT_FRAGMENT_NODE: 11
    }
}
```

17.1.6 Language-Independent DOM Interfaces

Although the DOM standard grew out of a desire to have a common API for dynamic HTML programming, the DOM is not of interest only to web scripters. In fact, the standard is currently most heavily used by server-side Java and C++ programs that parse and manipulate XML documents. Because of its many uses, the DOM standard is defined to be language-independent. This book describes only the JavaScript binding of the DOM

API, but you should be aware of a few other points. First, note that object properties in the JavaScript binding are typically mapped to pairs of get/set methods in other language bindings. Thus, when a Java programmer asks you about the `getFirstChild()` method of the Node interface, you need to understand that the JavaScript binding of the Node API doesn't define a `getFirstChild()` method. Instead, it simply defines a `firstChild` property, and reading the value of this property in JavaScript is equal to calling `getFirstChild()` in Java.

Another important feature of the JavaScript binding of the DOM API is that certain DOM objects behave like JavaScript arrays. If an interface defines a method named `item()`, objects that implement that interface behave like read-only numerical arrays. For example, suppose you've obtained a `NodeList` object by reading the `childNodes` property of a node. You can obtain the individual Node objects in the list by passing the desired node number to the `item()` method, or, more simply, you can simply treat the `NodeList` object as an array and index it directly. The following code illustrates these two options:

```
var n = document.documentElement; // This is a Node object.
var children = n.childNodes;     // This is a NodeList object.
var head = children.item(0);     // Here is one way to use a
NodeList.
var body = children[1];         // But this way is easier!
```

Similarly, if a DOM object has a `namedItem()` method, passing a string to this method is the same as using the string as an array index for the object. For example, the following lines of code are all equivalent ways to access a form element:

```
var f = document.forms.namedItem("myform");
var g = document.forms["myform"];
var h = document.forms.myform;
```

Because the DOM standard may be used in a variety of ways, the architects of the standard were careful to define the DOM API in a way that would not restrict the ability of others to implement the API as they saw fit. Specifically, the DOM standard defines interfaces instead of classes. In object-oriented programming, a class is a fixed data type that must be implemented exactly as specified. An interface, on the other hand, is a collection of methods and properties that must be implemented together. Thus, an implementation of the DOM is free to define whatever classes it sees fit, but those classes must define the methods and properties of the various DOM interfaces.

This architecture has a couple of important implications. First, the class names used in an implementation might not correspond directly to the interface names used in the DOM standard (and in this book). Second, a single class may implement more than one interface. For example, consider the `Document` object. This object is an instance of some class defined by the web browser implementation. We don't know what the specific class is, but we do know that it implements the `Document` interface; that is, all methods and properties defined by `Document` are available to us through the `Document` object. Since

web browsers work with HTML documents, we also know that the Document object implements the HTMLDocument interface and that all methods and properties defined by that interface are available to us as well. Furthermore, if a web browser supports CSS style sheets and implements the DOM CSS module, the Document object also implements the DocumentStyle and DocumentCSS DOM interfaces. And if the web browser supports the Events and Views modules, Document implements the DocumentEvent and DocumentView interfaces as well.

Because the DOM is broken into independent modules, it defines a number of minor add-on interfaces, such as DocumentStyle, DocumentEvent, and DocumentView, that define only one or two methods each. Interfaces such as these are never implemented independently of the core Document interface, and for this reason, I do not document them independently. When you look up the Document interface in the DOM reference section, you'll find that it also lists the methods and properties of its various add-on interfaces. Similarly, if you look up one of the add-on interfaces, you'll simply find a cross-reference to the core interface with which it is associated. The exception to this rule is when the add-on interface is a complex one. For example, the HTMLDocument interface is always implemented by the same object that implements the Document object, but because it adds substantial new functionality, I have given it a reference page of its own.

Another important fact you need to understand is that since the DOM standard defines interfaces instead of classes, it does not define any constructor methods. If you want to create a new Text object to insert into a document, for example, you cannot simply say:

```
var t = new Text("this is a new text node"); // No such constructor!
```

Since it cannot define constructors, the DOM standard instead defines a number of useful *factory methods* for creating objects in the Document interface. So, to create a new Text node for a document, you would write the following:

```
var t = document.createTextNode("this is a new text node");
```

Factory methods defined by the DOM have names that begin with the word "create". In addition to the factory methods defined by Document, a few others are defined by DOMImplementation and available as `document.implementation`.

17.2 Using the Core DOM API

Now that we've studied the tree structure of documents and seen how the tree is composed of Node objects, we can move on to study the Node object and document trees in more detail. As I noted previously, the core DOM API is not terribly complex. The following sections contain examples that demonstrate how you can use it to accomplish common tasks.

17.2.1 Traversing a Document

As we've already discussed, the DOM represents an HTML document as a tree of Node objects. With any tree structure, one of the most common things to do is traverse the tree, examining each node of the tree in turn. [Example 17-1](#) shows one way to do this. It is a JavaScript function that recursively examines a node and all its children, adding up the number of HTML tags (i.e., Element nodes) it encounters in the course of the traversal. Note the use of the `childNodes` property of a node. The value of this property is a NodeList object, which behaves (in JavaScript) like an array of Node objects. Thus, the function can enumerate all the children of a given node by looping through the elements of the `childNodes[]` array. By recursing, the function enumerates not just all children of a given node, but all nodes in the tree of nodes. Note that this function also demonstrates the use of the `nodeType` property to determine the type of each node.

Example 17-1. Traversing the nodes of a document

```
<head>
<script>
// This function is passed a DOM Node object and checks to see if that
node
// represents an HTML tag; i.e., if the node is an Element object. It
// recursively calls itself on each of the children of the node,
testing
// them in the same way. It returns the total number of Element objects
// it encounters. If you invoke this function by passing it the
// Document object, it traverses the entire DOM tree.
function countTags(n) { // n is a Node
    var numtags = 0; // Initialize the tag
counter
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // Check if n is an
Element
        numtags++; // Increment the
counter if so
    var children = n.childNodes; // Now get all children
of n
    for(var i=0; i < children.length; i++) { // Loop through the
children
        numtags += countTags(children[i]); // Recurse on each one
    }
    return numtags; // Return the total
number of tags
}
</script>
</head>
<!-- Here's an example of how the countTags( ) function might be used
-->
<body onload="alert('This document has ' + countTags(document) + '
tags')">
This is a <i>sample</i> document.
</body>
```

Another point to notice about [Example 17-1](#) is that the `countTags()` function it defines is invoked from the `onload` event handler, so that it is not called until the document is completely loaded. This is a general requirement when working with the DOM: you cannot traverse or manipulate the document tree until the document has been fully loaded.

In addition to the `childNodes` property, the Node interface defines a few other useful properties. `firstChild` and `lastChild` refer to the first and last children of a node, and `nextSibling` and `previousSibling` refer to adjacent siblings of a node. (Two nodes are siblings if they have the same parent node.) These properties provide another way to loop through the children of a node, demonstrated in [Example 17-2](#). This example counts the number of characters in all the Text nodes within the `<body>` of the document. Notice the way the `countCharacters()` function uses the `firstChild` and `nextSibling` properties to loop through the children of a node.

Example 17-2. Another way to traverse a document

```
<head>
<script>
// This function is passed a DOM Node object and checks to see if that
node
// represents a string of text; i.e., if the node is a Text object. If
// so, it returns the length of the string. If not, it recursively
calls
// itself on each of the children of the node and adds up the total
length
// of the text it finds. Note that it enumerates the children of a node
// using the firstChild and nextSibling properties. Note also that the
// function does not recurse when it finds a Text node, because Text
nodes
// never have children.
function countCharacters(n) { // n is a Node
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) // Check if n is a Text
object
        return n.length; // If so, return its
length
    // Otherwise, n may have children whose characters we need to count
    var numchars = 0; // Used to hold total characters of the children
    // Instead of using the childNodes property, this loop examines the
    // children of n using the firstChild and nextSibling properties.
    for(var m = n.firstChild; m != null; m = m.nextSibling) {
        numchars += countCharacters(m); // Add up total characters
found
    }
    return numchars; // Return total characters
}
</script>
</head>
<!--
    The onload event handler is an example of how the countCharacters( )
    function might be used. Note that we want to count only the
characters
    in the <body> of the document, so we pass document.body to the
function.
```

```
-->
<body onload="alert('Document length: ' +
countCharacters(document.body))">
This is a sample document.<p>How long is it?
</body>
```

17.2.2 Finding Specific Elements in a Document

The ability to traverse all nodes in a document tree gives us the power to find specific nodes. When programming with the DOM API, it is quite common to need a particular node within the document or a list of nodes of a specific type within the document. Fortunately, the DOM API provides functions that make this easy for us.

In [Example 17-2](#), we referred to the `<body>` element of an HTML document with the JavaScript expression `document.body`. The `body` property of the Document object is a convenient special-case property and is the preferred way to refer to the `<body>` tag of an HTML document. If this convenience property did not exist, however, we could also refer to the `<body>` tag like this:

```
document.getElementsByTagName("body")[0]
```

This expression calls the Document object's `getElementsByTagName()` method and selects the first element of the returned array. The call to `getElementsByTagName()` returns an array of all `<body>` elements within the document. Since HTML documents can have only one `<body>`, we know that we're interested in the first element of the returned array.^[6]

^[6] Technically, the DOM API specifies that `getElementsByTagName()` returns a `NodeList` object. In the JavaScript binding of the DOM, `NodeList` objects behave like arrays and are typically used that way.

You can use `getElementsByTagName()` to obtain a list of any type of HTML element. For example, to find all the tables within a document, you'd do this:

```
var tables = document.getElementsByTagName("table");
alert("This document contains " + tables.length + " tables");
```

Note that since HTML tags are not case-sensitive, the strings passed to `getElementsByTagName()` are also not case-sensitive. That is, the previous code finds `<table>` tags even if they are coded as `<TABLE>`. `getElementsByTagName()` returns elements in the order in which they appear in the document. Finally, if you pass the special string `"*"` to `getElementsByTagName()`, it returns a list of all the elements in the document, in the order in which they appear. (This special usage is not supported in IE 5 and IE 5.5. See instead the IE-specific `Document.all[]` array in the client-side reference section.)

Sometimes you don't want a list of elements but instead want to operate on a single specific element of a document. If you know a lot about the structure of the document, you may be able to use `getElementsByTagName()`. For example, if you want to do something to the fourth paragraph in a document, you might use this code:

```
var myParagraph = document.getElementsByTagName("p")[3];
```

This typically is not the best (nor the most efficient) technique, however, because it depends so heavily on the structure of the document; a new paragraph inserted at the beginning of the document would break the code. Instead, when you need to manipulate specific elements of a document, it is best to give those elements an `id` attribute that specifies a unique (within the document) name for the element. Then you can look up your desired element by its ID. For example, you might code the special fourth paragraph of your document with a tag like this:

```
<p id="specialParagraph">
```

You can then look up the node for that paragraph with JavaScript code like this:

```
var myParagraph = document.getElementById("specialParagraph");
```

Note that the `getElementById()` method does not return an array of elements like `getElementsByTagName()` does. Because the value of every `id` attribute is (or is supposed to be) unique, `getElementById()` returns only the single element with the matching `id` attribute. `getElementById()` is an important method, and its use is quite common in DOM programming.

`getElementById()` and `getElementsByTagName()` are both methods of the Document object. Element objects also define a `getElementsByTagName()` method, however. This method of the Element object behaves just like the method of the Document object, except that it returns only elements that are descendants of the element on which it is invoked. Instead of searching the entire document for elements of a specific type, it searches only within the given element. This makes it possible, for example, to use `getElementById()` to find a specific element and then to use `getElementsByTagName()` to find all descendants of a given type within that specific tag. For example:

```
// Find a specific Table element within a document and count its rows
var tableOfContents = document.getElementById("TOC");
var rows = tableOfContents.getElementsByTagName("tr");
var numRows = rows.length;
```

Finally, note that for HTML documents, the `HTMLDocument` object also defines a `getElementsByName()` method. This method is like `getElementById()`, but it looks at the `name` attribute of elements rather than the `id` attribute. Also, because the `name` attribute is not expected to be unique within a document (for example, radio buttons within HTML forms usually have the same `name`), `getElementsByName()` returns an array of elements rather than a single element. For example:

```
// Find <a name="top">
var link = document.getElementsByName("top")[0];
// Find all <input type="radio" name="shippingMethod"> elements
var choices = document.getElementsByName("shippingMethod");
```

17.2.3 Modifying a Document

Traversing the nodes of a document can be useful, but the real power of the core DOM API lies in the features that allow you to use JavaScript to dynamically modify documents. The following examples demonstrate the basic techniques of modifying documents and illustrate some of the possibilities.

[Example 17-3](#) includes a JavaScript function named `reverse()`, a sample document, and an HTML button that, when pressed, calls the `reverse()` function, passing it the node that represents the `<body>` element of the document. (Note the use of `getElementsByTagName()` within the button's event handler to find the `<body>` element.) The `reverse()` function loops backward through the children of the supplied node and uses the `removeChild()` and `appendChild()` methods of the `Node` object to reverse the order of those children.

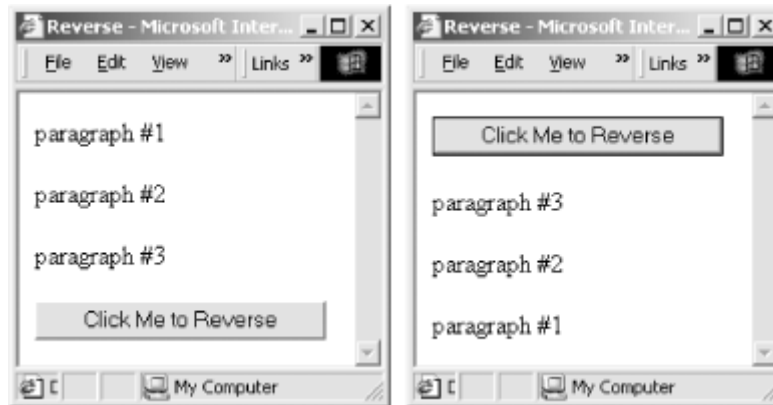
Example 17-3. Reversing the nodes of a document

```
<head><title>Reverse</title>
<script>
function reverse(n) {           // Reverse the order of the children of
Node n
    var kids = n.childNodes;    // Get the list of children
    var numkids = kids.length; // Figure out how many children there
are
    for(var i = numkids-1; i >= 0; i--) { // Loop backward through the
children
        var c = n.removeChild(kids[i]); // Remove a child
        n.appendChild(c);             // Put it back at its new
position
    }
}
</script>
</head>
<body>
<p>paragraph #1<p>paragraph #2<p>paragraph #3  <!-- A sample document -
->
<p>                                     <!-- A button to call reverse(
)-->
<button onclick="reverse(document.body);"
```

```
>Click Me to Reverse</button>
</body>
```

The result of [Example 17-3](#), illustrated in [Figure 17-3](#), is that when the user clicks the button, the order of the paragraphs and of the button are reversed.

Figure 17-3. A document before and after being reversed



There are a couple of points worth noting about [Example 17-3](#). First, if you pass a node that is already part of the document to `appendChild()` it first removes it, so we could have simplified our code by omitting the call to `removeChild()`. Second, keep in mind that the `childNodes` property (like all `NodeList` objects) is "live": when the document is modified, the modifications are immediately visible through the `NodeList`. This is an important feature of the `NodeList` interface, but it can actually make some code trickier to write. A call to `removeChild()`, for example, changes the index of all the siblings that follow that child, so if you want to iterate through a `NodeList` and delete some of the nodes, you must write your looping code carefully.

[Example 17-4](#) shows a variation on the `reverse()` function of the previous example. This one uses recursion to reverse not only the children of a specified node, but also all the node's descendants. In addition, when it encounters a `Text` node, it reverses the order of the characters within that node. [Example 17-4](#) shows only the JavaScript code for this new `reverse()` function. It could easily be used in an HTML document like the one shown in [Example 17-3](#), however.

Example 17-4. A recursive node-reversal function

```
// Recursively reverse all nodes beneath Node n and reverse Text nodes
function reverse(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) { // Reverse Text nodes
        var text = n.data; // Get content of node
        var reversed = "";
        for(var i = text.length-1; i >= 0; i--) // Reverse it
            reversed += text.charAt(i);
        n.data = reversed; // Store reversed text
    }
}
```

```

    else { // For non-Text nodes, recursively reverse the order of the
children
        var kids = n.childNodes;
        var numkids = kids.length;
        for(var i = numkids-1; i >= 0; i--) { // Loop through
kids
            reverse(kids[i]); // Recurse to
reverse kid
            n.appendChild(n.removeChild(kids[i])); // Move kid to new
position
        }
    }
}

```

[Example 17-4](#) shows one way to change the text displayed in a document: simply set the `data` field of the appropriate Text node. [Example 17-5](#) shows another way. This example defines a function, `uppercase()`, that recursively traverses the children of a given node. When it finds a Text node, the function replaces that node with a new Text node containing the text of the original node, converted to uppercase. Note the use of the `document.createTextNode()` method to create the new Text node and the use of Node's `replaceChild()` method to replace the original Text node with the newly created one. Note also that `replaceChild()` is invoked on the parent of the node to be replaced, not on the node itself. The `uppercase()` function uses Node's `parentNode` property to determine the parent of the Text node it replaces.

In addition to defining the `uppercase()` function, [Example 17-5](#) includes two HTML paragraphs and a button. When the user clicks the button, one of the paragraphs is converted to uppercase. Each paragraph is identified with a unique name, specified with the `id` attribute of the `<p>` tag. The event handler on the button uses the `getElementById()` method to get the Element object that represents the desired paragraph.

Example 17-5. Replacing nodes with their uppercase equivalents

```

<script>
// This function recursively looks at Node n and its descendants,
// replacing all Text nodes with their uppercase equivalents.
function uppercase(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {
        // If the node is a Text node, create a new Text node that
        // holds the uppercase version of the node's text, and use the
        // replaceChild( ) method of the parent node to replace the
        // original node with the new uppercase node.
        var newNode = document.createTextNode(n.data.toUpperCase( ));
        var parent = n.parentNode;
        parent.replaceChild(newNode, n);
    }
    else {
        // If the node is not a Text node, loop through its children
        // and recursively call this function on each child.
        var kids = n.childNodes;

```

```

        for(var i = 0; i < kids.length; i++) uppercase(kids[i]);
    }
}
</script>

<!-- Here is some sample text. Note that the <p> tags have id
attributes. -->
<p id="p1">This <i>is</i> paragraph 1.</p>
<p id="p2">This <i>is</i> paragraph 2.</p>

<!-- Here is a button that invokes the uppercase( ) function defined
above. -->
<!-- Note the call to document.getElementById( ) to find the desired
node. -->
<button onclick="uppercase(document.getElementById('p1'));">Click
Me</button>

```

The previous two examples show how to modify document content by replacing the text contained within a Text node and by replacing one Text node with an entirely new Text node. It is also possible to append, insert, delete, or replace text within a Text node with the `appendData()`, `insertData()`, `deleteData()`, and `replaceData()` methods. These methods are not directly defined by the Text interface, but instead are inherited by Text from CharacterData. You can find more information about them under "CharacterData" in the DOM reference section.

In the node-reversal examples, we saw how we could use the `removeChild()` and `appendChild()` methods to reorder the children of a Node. Note, however, that we are not restricted to changing the order of nodes within their parent node; the DOM API allows nodes in the document tree to be moved freely within the tree (only within the same document, however). [Example 17-6](#) demonstrates this by defining a function named `embolden()` that replaces a specified node with a new element (created with the `createElement()` method of the Document object) that represents an HTML `` tag and "reparents" the original node as a child of the new `` node. In an HTML document, this causes any text within the node or its descendants to be displayed in boldface.

Example 17-6. Reparenting a node to a `` element

```

<script>
// This function takes a Node n, replaces it in the tree with an
Element node
// that represents an HTML <b> tag, and then makes the original node
the
// child of the new <b> element.
function embolden(node) {
    var bold = document.createElement("b"); // Create a new <b>
element
    var parent = node.parentNode;           // Get the parent of the
node
    parent.replaceChild(bold, node);        // Replace the node with
the <b> tag
    bold.appendChild(node);                 // Make the node a child
of the <b> tag

```

```

}
</script>

<!-- A couple of sample paragraphs -->
<p id="p1">This <i>is</i> paragraph #1.</p>
<p id="p2">This <i>is</i> paragraph #2.</p>

<!-- A button that invokes the embolden( ) function on the first
paragraph -->
<button
onclick="embolden(document.getElementById('p1'));">Embalden</button>

```

In addition to modifying documents by inserting, deleting, reparenting, and otherwise rearranging nodes, it is also possible to make substantial changes to a document simply by setting attribute values on document elements. One way to do this is with the `element.setAttribute()` method. For example:

```

var headline = document.getElementById("headline"); // Find named
element
headline.setAttribute("align", "center"); // Set
align='center'

```

The DOM elements that represent HTML attributes define JavaScript properties that correspond to each of their standard attributes (even deprecated attributes such as `align`), so you can also achieve the same effect with this code:

```

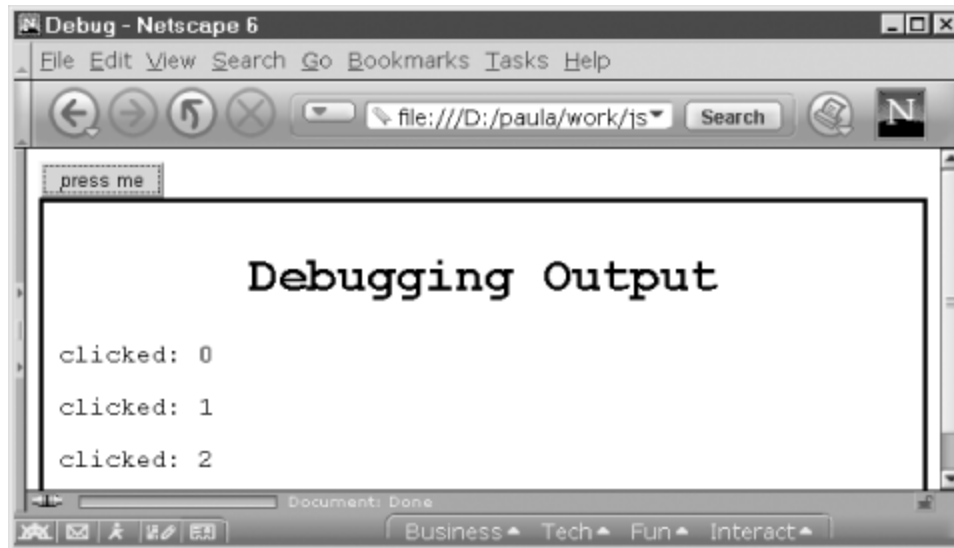
var headline = document.getElementById("headline");
headline.align = "center"; // Set alignment attribute.

```

17.2.4 Adding Content to a Document

The previous two examples showed how the contents of a Text node can be changed to uppercase and how a node can be reparented to be a child of a newly created `` node. The `embolden()` function showed that it is possible to create new nodes and add them to a document. You can add arbitrary content to a document by creating the necessary Element nodes and Text nodes with `document.createElement()` and `document.createTextNode()` and by adding them appropriately to the document. This is demonstrated in [Example 17-7](#), which defines a function named `debug()`. This function provides a convenient way to insert debugging messages into a program, and it serves as a useful alternative to using the built-in `alert()` function. A sample use of this `debug()` function is illustrated in [Figure 17-4](#).

Figure 17-4. Output of the `debug()` function



The first time `debug()` is called, it uses the DOM API to create a `<div>` element and insert it at the end of the document. The debugging messages passed to `debug()` on this first call and all subsequent calls are then inserted into this `<div>` element. Each debugging message is displayed by creating a Text node within a `<p>` element and inserting that `<p>` element at the end of the `<div>` element.

[Example 17-7](#) also demonstrates a convenient but nonstandard way to add new content to a document. The `<div>` element that contains the debugging messages displays a large, centered title. This title could be created and added to the document in the way that other content is, but in this example we instead use the `innerHTML` property of the `<div>` element. Setting this property of any element to a string of HTML text causes that HTML to be parsed and inserted as the content of the element. Although this property is not part of the DOM API, it is a useful shortcut that is supported by Internet Explorer 4 and later and Netscape 6. Although it is not standard, it is in common use and is included in this example for completeness.^[7]

^[7] The `innerHTML` property is particularly useful when you want to insert large or complex chunks of HTML text into a document. For simple fragments of HTML, using DOM methods is more efficient because no HTML parser is required. Note that appending bits of text to the `innerHTML` property with the `+=` operator is usually not efficient.

Example 17-7. Adding debugging output to a document

```
/**
 * This debug function displays plain-text debugging messages in a
 * special box at the end of a document. It is a useful alternative
 * to using alert( ) to display debugging messages.
 */
function debug(msg) {
    // If we haven't already created a box within which to display
    // our debugging messages, then do so now. Note that to avoid
    // using another global variable, we store the box node as
    // a property of this function.
    if (!debug.box) {
```

```

// Create a new <div> element
debug.box = document.createElement("div");
// Specify what it looks like using CSS style attributes
debug.box.setAttribute("style",
    "background-color: white; " +
    "font-family: monospace; " +
    "border: solid black 3px; " +
    "padding: 10px;");

// Append our new <div> element to the end of the document
document.body.appendChild(debug.box);

// Now add a title to our <div>. Note that the innerHTML
property is
// used to parse a fragment of HTML and insert it into the
document.
// innerHTML is not part of the W3C DOM standard, but it is
supported
// by Netscape 6 and Internet Explorer 4 and later. We can
avoid
// the use of innerHTML by explicitly creating the <h1>
element,
// setting its style attribute, adding a Text node to it, and
// inserting it into the document, but this is a nice shortcut.
debug.box.innerHTML = "<h1 style='text-align:center'>Debugging
Output</h1>";
}

// When we get here, debug.box refers to a <div> element into which
// we can insert our debugging message.
// First create a <p> node to hold the message.
var p = document.createElement("p");
// Now create a text node containing the message, and add it to the
<p>
p.appendChild(document.createTextNode(msg));
// And append the <p> node to the <div> that holds the debugging
output
debug.box.appendChild(p);
}

```

The `debug()` method listed in [Example 17-7](#) can be used in HTML documents like the following, which is the document that was used to generate [Figure 17-4](#):

```

<script src="Debug.js"></script> <!-- Include the debug( ) function -->
<script>var numtimes=0;</script> <!-- Define a global variable -->
<!-- Now use the debug( ) function in an event handler -->
<button onclick="debug('clicked: ' + numtimes++);">press me</button>

```


17.2.5 Working with Document Fragments

The core DOM API defines the `DocumentFragment` object as a convenient way of working with groups of Document nodes. A `DocumentFragment` is a special type of node that does not appear in a document itself but serves as a temporary container for a sequential collection of nodes and allows those nodes to be manipulated as a single object. When a `DocumentFragment` is inserted into a document (using any of the `appendChild()`, `insertBefore()`, or `replaceChild()` methods of the `Node` object), it is not the `DocumentFragment` itself that is inserted, but each of its children.

As an example, you can use a `DocumentFragment` to rewrite the `reverse()` method of [Example 17-3](#) like this:

```
function reverse(n) { // Reverses the order of the children of Node n
    var f = document.createDocumentFragment( ); // Get an empty
DocumentFragment
    while(n.lastChild) // Loop backward through the
children,
        f.appendChild(n.lastChild); // moving each one to the
DocumentFragment
    n.appendChild(f); // Then move them back (in their
new order)
}
```

Once you have created a `DocumentFragment`, you can use it with code like this:

```
document.getElementsByTagName("p")[0].appendChild(fragment);
```

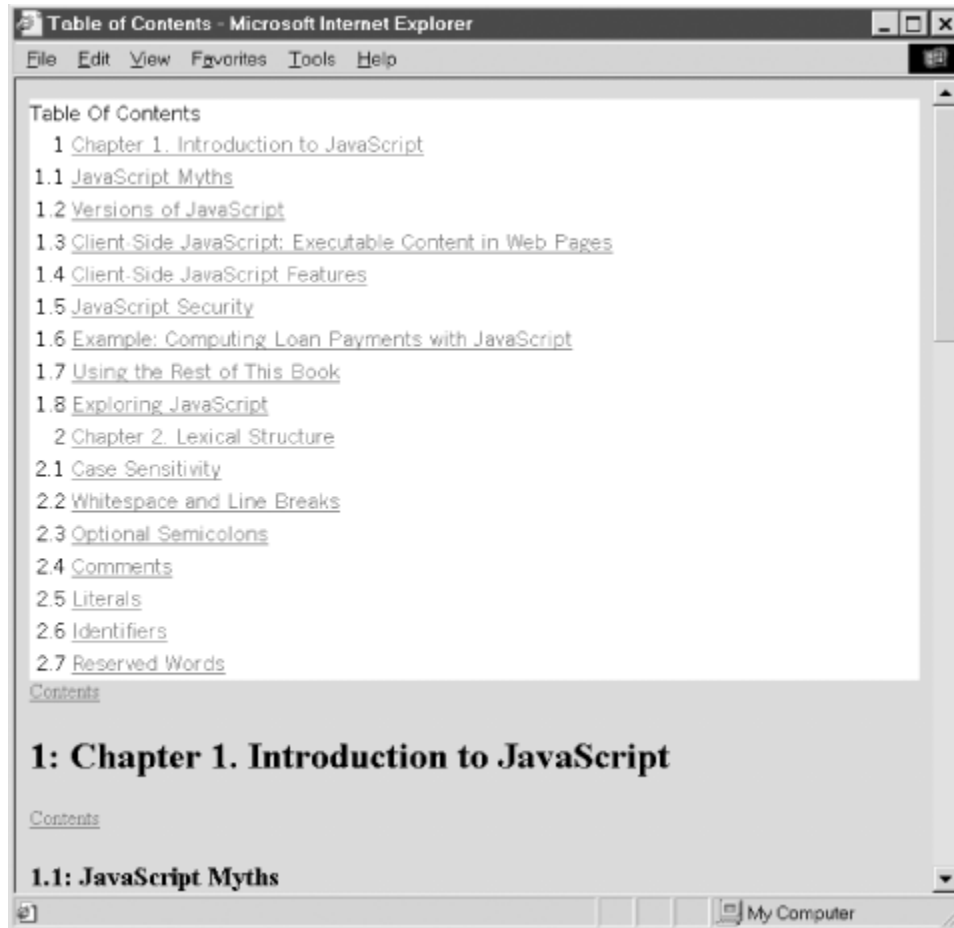
Note that when you insert a `DocumentFragment` into a document, the child nodes of the fragment are moved from the fragment into the document. After the insertion, the fragment is empty and cannot be reused unless you first add new children to it. We'll see the `DocumentFragment` object again later in this chapter, when we examine the DOM Range API.

17.2.6 Example: A Dynamically Created Table of Contents

The previous sections showed how you can use the core DOM API to traverse, modify, and add content to a document. [Example 17-8](#), at the end of this section, puts all these pieces together into a single longer example. The example defines a single method, `maketoc()`, which expects a `Document` node as its single argument. `maketoc()` traverses the document, creates a table of contents (TOC) for it, and replaces the specified node with the newly created TOC. The TOC is generated by looking for `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, and `<h6>` tags within the document and assuming that these tags mark the beginnings of important sections within the document. In addition to creating a TOC, the `maketoc()` function inserts named anchors (`<a>` elements with the `name` attribute set instead of the `href` attribute) before each section heading so that the TOC can link

directly to each section. Finally, `maketoc()` also inserts links at the beginning of each section back to the TOC; when the reader reaches a new section, she can either read that section or follow the link back to the TOC and choose a new section. [Figure 17-5](#) shows what a TOC generated by the `maketoc()` function looks like.

Figure 17-5. A dynamically created table of contents



If you maintain and revise long documents that are broken into sections with `<h1>`, `<h2>`, and related tags, the `maketoc()` function may be of interest to you. TOCs are quite useful in long documents, but when you frequently revise a document it can be difficult to keep the TOC in sync with the document itself. The TOC for this book was automatically created by postprocessing the content of the book. `maketoc()` allows you to do something similar for your web documents. You can use the function in an HTML document like this one:

```
<script src="TOC.js"></script> <!-- Load the maketoc( ) function -->
<!-- Call the maketoc( ) function when the document is fully loaded -->
>
<body onload="maketoc(document.getElementById('placeholder'))">
<!-- This span element will be replaced by the generated TOC -->
<span id="placeholder">Table Of Contents</span>
```

```
// ... rest of document goes here ...
```

Another way to use the `maketoc()` function is to generate the TOC only when the reader requests it. You can do this by including a link (or button) that replaces itself with the generated TOC when the user clicks on it:

```
<a href="#" onclick="maketoc(this); return false;">Show Table Of  
Contents</a>
```

The code for the `maketoc()` function follows. [Example 17-8](#) is long, but it is well commented and uses techniques that have already been demonstrated. It is worth studying as a practical example of the power of the DOM API. Note that the `maketoc()` function relies on two helper functions. For modularity, these helper functions are defined inside `maketoc()` itself. This prevents the addition of extra unnecessary functions to the global namespace.

Example 17-8. Automatically generating a table of contents

```
/**  
 * Create a table of contents for this document, and insert the TOC  
into  
 * the document by replacing the node specified by the replace  
argument.  
 **/  
function maketoc(replace) {  
    // Create a <div> element that is the root of the TOC tree  
    var toc = document.createElement("div");  
  
    // Set a background color and font for the TOC. We'll learn about  
    // the style property in the next chapter.  
    toc.style.backgroundColor = "white";  
    toc.style.fontFamily = "sans-serif";  
  
    // Start the TOC with an anchor so we can link back to it  
    var anchor = document.createElement("a"); // Create an <a> node  
    anchor.setAttribute("name", "TOC"); // Give it a name  
    toc.appendChild(anchor); // Insert it  
  
    // Make the body of the anchor the title of the TOC  
    anchor.appendChild(document.createTextNode("Table Of Contents"));  
  
    // Create a <table> element that will hold the TOC and add it  
    var table = document.createElement("table");  
    toc.appendChild(table);  
  
    // Create a <tbody> element that holds the rows of the TOC  
    var tbody = document.createElement("tbody");  
    table.appendChild(tbody);  
  
    // Initialize an array that keeps track of section numbers  
    var sectionNumbers = [0,0,0,0,0,0];
```

```

    // Recursively traverse the body of the document, looking for
sections
    // sections marked with <h1>, <h2>, ... tags, and use them to
create
    // the TOC by adding rows to the table
    addSections(document.body, tbody, sectionNumbers);

    // Finally, insert the TOC into the document by replacing the node
    // specified by the replace argument with the TOC subtree
    replace.parentNode.replaceChild(toc, replace);

    // This method recursively traverses the tree rooted at Node n,
looking
    // looking for <h1> through <h6> tags, and uses the content of
these tags
    // to build the table of contents by adding rows to the HTML table
specified
    // by the toc argument. It uses the sectionNumbers array to keep
track of
    // the current section number.
    // This function is defined inside of maketoc( ) so that it is not
    // visible from the outside. maketoc( ) is the only function
exported
    // by this JavaScript module.
    function addSections(n, toc, sectionNumbers) {
        // Loop through all the children of n
        for(var m = n.firstChild; m != null; m = m.nextSibling) {
            // Check whether m is a heading element. It would be nice
if we
            // could just use (m instanceof HTMLHeadingElement), but
this is
            // not required by the specification and it does not work
in IE.
            // Therefore, we must check the tagname to see if it is H1-
H6.
            if ((m.nodeType == 1) && /* Node.ELEMENT_NODE */
                (m.tagName.length == 2) && (m.tagName.charAt(0) ==
"H")) {
                // Figure out what level heading it is
                var level = parseInt(m.tagName.charAt(1));
                if (!isNaN(level) && (level >= 1) && (level <= 6)) {
                    // Increment the section number for this heading
level
                    sectionNumbers[level-1]++;
                    // And reset all lower heading-level numbers to
zero
                    for(var i = level; i < 6; i++) sectionNumbers[i] =
0;
                    // Now combine section numbers for all heading
levels
                    // to produce a section number like "2.3.1"
                    var sectionNumber = "";
                    for(var i = 0; i < level; i++) {
                        sectionNumber += sectionNumbers[i];
                        if (i < level-1) sectionNumber += ".";
                    }
                }
            }
        }
    }

```

```

        // Create an anchor to mark the beginning of this
section
        // This will be the target of a link we add to the
TOC
        var anchor = document.createElement("a");
        anchor.setAttribute("name", "SECT"+sectionNumber);

        // Create a link back to the TOC and make it a
        // child of the anchor
        var backlink = document.createElement("a");
        backlink.setAttribute("href", "#TOC");

backlink.appendChild(document.createTextNode("Contents"));
        anchor.appendChild(backlink);

        // Insert the anchor into the document right before
the
        // section header
        n.insertBefore(anchor, m);

        // Now create a link to this section. It will be
added
        // to the TOC below.
        var link = document.createElement("a");
        link.setAttribute("href", "#SECT" + sectionNumber);
        // Get the heading text using a function defined
below
        var sectionTitle = getTextContent(m);
        // Use the heading text as the content of the link

link.appendChild(document.createTextNode(sectionTitle));

        // Create a new row for the TOC
        var row = document.createElement("tr");
        // Create two columns for the row
        var col1 = document.createElement("td");
        var col2 = document.createElement("td");
        // Make the first column right-aligned and put the
section
        // number in it
        col1.setAttribute("align", "right");

col1.appendChild(document.createTextNode(sectionNumber));
        // Put a link to the section in the second column
col2.appendChild(link);
        // Add the columns to the row, and the row to the
table
        row.appendChild(col1);
        row.appendChild(col2);
        toc.appendChild(row);

        // Modify the section header element itself to add
        // the section number as part of the section title
m.insertBefore(document.createTextNode(sectionNumber+": "),
                m.firstChild);
    }

```

```

    }
    else { // Otherwise, this is not a heading element, so
recurse
        addSections(m, toc, sectionNumbers);
    }
}
}

// This utility function traverses Node n, returning the content of
// all Text nodes found and discarding any HTML tags. This is also
// defined as a nested function, so it is private to this module.
function getTextContent(n) {
    var s = '';
    var children = n.childNodes;
    for(var i = 0; i < children.length; i++) {
        var child = children[i];
        if (child.nodeType == 3 /*Node.TEXT_NODE*/) s +=
child.data;
        else s += getTextContent(child);
    }
    return s;
}
}
}

```

17.2.7 Working with XML Documents

Web browsers display HTML documents, but XML documents are becoming more and more important as sources of data. Since the DOM allows us to traverse and manipulate both HTML and XML documents, we can use DOM methods to load an XML document, extract information from it, and dynamically create an HTML version of that information for display in a web browser. [Example 17-9](#) shows how this can be done in Netscape 6.1 and Internet Explorer 6. It is an HTML file that consists mostly of JavaScript code. The file expects to be loaded through a URL that uses the URL query string to specify the relative URL of the data file to load. For example, you might invoke this example file with a URL like this:

```
file:///C:/javascript/DisplayEmployeeData.html?data.xml
```

DisplayEmployeeData.html is the name of the example file, and *data.xml* is the name of the XML file it uses. The XML file must contain data formatted like this:

```

<employees>
  <employee name="J.
Doe"><job>Programmer</job><salary>32768</salary></employee>
  <employee name="A.
Baker"><job>Sales</job><salary>70000</salary></employee>
  <employee name="Big
Cheese"><job>CEO</job><salary>1000000</salary></employee>
</employees>

```

The example contains two JavaScript functions. The first, `loadXML()`, is a generic function for loading any XML file. It contains standard DOM Level 2 code to load the XML document and also code that uses a proprietary Microsoft API to accomplish the same thing. The only really new thing in this example is the creation of a new Document object with the `DOMImplementation.createDocument()` method and the call to the `load()` method of that Document object. An important thing to notice here is that documents do not load instantaneously, so the call to `loadXML()` returns before the document is loaded. For this reason, we pass `loadXML()` a reference to another function that it should call when the document has finished loading.

The other function in the example is `makeTable()`. This is the function that we pass to `loadXML()`. When the XML file finishes loading, it passes the Document object representing the XML file and the URL of the file to `makeTable()`. `makeTable()` uses DOM methods we've seen before to extract information from the XML document and insert it into a table in the HTML document displayed by the browser. This function also illustrates the use of some table-related convenience methods defined by `HTMLTableElement`, `HTMLTableRowElement`, and related interfaces. See the DOM reference section for complete details about these table-specific interfaces and their methods. Although the DOM methods and properties used in this function are all straightforward, they are used in dense combinations. Study the code carefully and you should have no difficulty understanding it.

Example 17-9. Loading and reading data from an XML document

```
<head><title>Employee Data</title>
<script>
// This function loads the XML document from the specified URL and,
when
// it is fully loaded, passes that document and the URL to the
specified
// handler function. This function works with any XML document.
function loadXML(url, handler) {
    // Use the standard DOM Level 2 technique, if it is supported
    if (document.implementation &&
document.implementation.createDocument) {
        // Create a new Document object
        var xmldoc = document.implementation.createDocument("", "",
null);
        // Specify what should happen when it finishes loading
        xmldoc.onload = function( ) { handler(xmldoc, url); }
        // And tell it what URL to load
        xmldoc.load(url);
    }
    // Otherwise, use Microsoft's proprietary API for Internet Explorer
    else if (window.ActiveXObject) {
        var xmldoc = new ActiveXObject("Microsoft.XMLDOM"); // Create
doc
        xmldoc.onreadystatechange = function( ) { //
Specify onload
            if (xmldoc.readyState == 4) handler(xmldoc, url);
        }
    }
}
```

```

        xmlDoc.load(url); // Start
loading!
    }
}

// This function builds an HTML table of employees from data it reads
from
// the XML document it is passed
function makeTable(xmlDoc, url) {
    // Create a <table> object and insert it into the document
    var table = document.createElement("table");
    table.setAttribute("border", "1");
    document.body.appendChild(table);

    // Use convenience methods of HTMLTableElement and related
interfaces
    // to define a table caption and a header that gives a name to each
column
    var caption = "Employee Data from " + url;
    table.createCaption(
).appendChild(document.createTextNode(caption));
    var header = table.createTHead( );
    var headerrow = header.insertRow(0);

headerrow.insertCell(0).appendChild(document.createTextNode("Name"));

headerrow.insertCell(1).appendChild(document.createTextNode("Job"));

headerrow.insertCell(2).appendChild(document.createTextNode("Salary"));

    // Now find all <employee> elements in our xmlDoc document
    var employees = xmlDoc.getElementsByTagName("employee");

    // Loop through these <employee> elements
    for(var i = 0; i < employees.length; i++) {
        // For each employee, get name, job, and salary data using
standard DOM
        // methods. The name comes from an attribute. The other values
are
        // in Text nodes within <job> and <salary> tags.
        var e = employees[i];
        var name = e.getAttribute("name");
        var job = e.getElementsByTagName("job")[0].firstChild.data;
        var salary =
e.getElementsByTagName("salary")[0].firstChild.data;

        // Now that we have the employee data, use methods of the table
to
        // create a new row and then use the methods of the row to
create
        // new cells containing the data as Text nodes
        var row = table.insertRow(i+1);
        row.insertCell(0).appendChild(document.createTextNode(name));
        row.insertCell(1).appendChild(document.createTextNode(job));
        row.insertCell(2).appendChild(document.createTextNode(salary));
    }
}

```



```
</script>
</head>
<!--
The body of the document contains no static text; everything is
dynamically
generated by the makeTable( ) function. The onload event handler
starts
things off by calling loadXML( ) to load the XML data file. Note the
use of
location.search to encode the name of the XML file in the query string.
Load
this HTML file with a URL like this: DisplayEmployeeData.html?data.xml.
-->
<body onload="loadXML(location.search.substring(1), makeTable)">
</body>
```

17.3 DOM Compatibility with Internet Explorer 4

Although IE 4 is not DOM-compliant, it has features that are similar to the core DOM APIs. These features are not part of the DOM standard and are not compatible with Netscape, but they are compatible with later versions of IE. The features are summarized here; consult the client-side reference section of this book for more details.

17.3.1 Traversing a Document

The DOM standard specifies that all Node objects, which includes both the Document object and all Element objects, have a `childNodes[]` array that contains the children of that node. IE 4 does not support `childNodes[]`, but it provides a very similar `children[]` array on its Document and HTML Element objects. Thus, it is easy to write a recursive function like the one shown in [Example 17-1](#) to traverse the complete set of HTML elements within an IE 4 document.

There is one substantial difference between IE 4's `children[]` array and the standard DOM `childNodes[]` array, however. IE 4 does not have a Text node type and does not consider strings of text to be children. Thus, a `<p>` tag that contains only plain text with no markup has an empty `children[]` array in IE 4. As we'll see shortly, however, the textual content of a `<p>` tag is available through the IE 4 `innerText` property.

17.3.2 Finding Document Elements

IE 4 does not support the `getElementById()` and `getElementsByTagName()` methods of the Document object. Instead, the Document object and all document elements have an array property named `all[]`. As the name suggests, this array represents *all* the elements in a document or all the elements contained within another element. Note that `all[]` does not simply represent the children of the document or the element -- it represents all descendants, no matter how deeply nested.

The `all[]` array can be used in several ways. If you index it with an integer n , it returns the $n+1$ th element of the document or the parent element. For example:

```
var e1 = document.all[0]; // The first element of the document
var e2 = e1.all[4];      // The fifth element of element 1
```

Elements are numbered in the order in which they appear in the document source. Note the one big difference between the IE 4 API and the DOM standard: IE does not have a notion of Text nodes, so the `all[]` array contains only document elements, not the text that appears within them.

It is usually much more useful to be able to refer to document elements by name rather than number. The IE 4 equivalent to `getElementById()` is to index the `all[]` array with a string rather than a number. When you do this, IE 4 returns the element whose `id` or `name` attribute has the specified value. If there is more than one such element (which can happen, since it is common to have multiple form elements, such as radioboxes, with the same `name` attribute), the result is an array of those elements. For example:

```
var specialParagraph = document.all["special"];
var radioboxes = form.all["shippingMethod"]; // May return an array
```

JavaScript also allows us to write these expressions by expressing the array index as a property name:

```
var specialParagraph = document.all.special;
var radioboxes = form.all.shippingMethod;
```

Using the `all[]` array in this way provides the same basic functionality as `getElementById()` and `getElementsByName()`. The main difference is that the `all[]` array combines the features of these two methods, which can cause problems if you inadvertently use the same values for the `id` and `name` attributes of unrelated elements.

The `all[]` array has an unusual quirk: a `tags()` method that can be used to obtain an array of elements by tag name. For example:

```
var lists = document.all.tags("UL"); // Find all <ul> tags in the
document
var items = lists[0].all.tags("LI"); // Find all <li> tags in the
first <ul>
```

This IE 4 syntax provides essentially the same functionality as the DOM Document and Element objects' `getElementsByTagName()` method. Note that in IE 4, the tag name should be specified using all capital letters.

17.3.3 Modifying Documents

Like the DOM standard, IE 4 exposes the attributes of HTML tags as properties of the corresponding HTML`Element` objects. Thus, it is possible to modify a document displayed in IE 4 by dynamically changing its HTML attributes. If an attribute modification changes the size of any element, the document "reflows" to accommodate its new size. The IE 4 HTML`Element` object defines `setAttribute()`, `getAttribute()`, and `removeAttribute()` methods as well. These are similar to the methods of the same name defined by the `Element` object in the standard DOM API.

The DOM standard defines an API that makes it possible to create new nodes, insert nodes into the document tree, reparent nodes, and move nodes within the tree. IE 4 cannot do this. Instead, however, all HTML`Element` objects in IE 4 define an `innerHTML` property. Setting this property to a string of HTML text allows you to replace the content of an element with whatever you want. Because this `innerHTML` property is so powerful, it has been implemented by Netscape 6 (and the Mozilla browser from which it is derived), even though it is not part of the DOM standard. `innerHTML` was demonstrated in [Example 17-7](#).

IE 4 also defines several related properties and methods. The `outerHTML` property replaces an element's content and the entire element itself with a specified string of HTML text. The `innerText` and `outerText` properties are similar to `innerHTML` and `outerHTML`, except that they treat the string as plain text and do not parse it as HTML. Finally, the `insertAdjacentHTML()` and `insertAdjacentText()` methods leave the content of an element alone but insert new HTML or plain-text content near (before or after, inside or outside) it. These properties and functions are not as commonly used as `innerHTML` and have not been implemented by Netscape 6. For further details, see "HTML`Element`" in the client-side reference section.

17.4 DOM Compatibility with Netscape 4

Netscape 4 does not even come close to implementing the DOM standard. In particular, Netscape 4 provides no way to access or set attributes on arbitrary elements of a document. Netscape 4 supports the Level 0 DOM API, of course, so elements such as forms and links can be accessed through the `forms[]` and `links[]` arrays, but there is no general way to traverse the children of these elements or set arbitrary attributes on them. Furthermore, Netscape 4 does not have the ability to "reflow" document content in response to changes in element size.

Despite these restrictions, Netscape 4 does provide an API that allows access to and manipulation of the crucial "dynamic elements" used to implement DHTML effects. In the Netscape 4 API, these elements are known as *layers*; they float above the rest of the document and can be moved, resized, and modified independently of the other elements of the document. Layers are typically implemented using CSS style sheets, and the Netscape 4 Layer API is discussed in detail in [Chapter 18](#).

What follows is simply an overview that explains how you can create, access, and modify the content of individual layer elements within a document. Although Netscape 4 does not support anything like the DOM standard, its Layer API allows you to achieve some of the same dynamic effects that are possible with the standard API. Note that the Layer API was submitted to the W3C for consideration as part of the DOM standard, but no part of this API was ever standardized. Because Netscape 6 is based on a complete rewrite of Netscape 4, the Layer API has been abandoned and is not supported in Netscape 6 (or in Mozilla).

Layers can be created in a document using the `<layer>` tag, a proprietary Netscape extension to HTML. More commonly, however, you create a layer in a Netscape 4 document using standard CSS positioning attributes (which will be explained in detail in [Chapter 18](#)). Any element made dynamic with CSS style attributes is treated as a layer by Netscape 4 and can be manipulated using the Layer API. (Note, though, that Netscape 4 does not allow all elements to be made dynamic. To be safe, a `<div>` wrapper element is usually used around any element that is to be dynamic.) JavaScript can also dynamically create layers using the `Layer()` constructor, which you can read about in the client-side reference section of this book.

Once you've created dynamic elements, or layers, in your document, Netscape 4 allows you to access them via a simple extension of the Level 0 DOM API. Just as you access form elements through a `forms[]` array and image elements through an `images[]` array, so do you access layers through a `layers[]` array of the Document object. If the first layer in a document has a `name` attribute of "layer1", you can refer to that layer element with any of the following expressions:

```
document.layers[0]           // Index the array with a number
document.layers['layer1']    // Index the array with an element name
document.layer1              // Named layers become a document property
```

If a layer has no `name` attribute but has an `id` attribute, the value of this attribute is used as the layer name instead.

Layers in your documents are represented by Layer objects that define a number of useful properties and methods you can use to move, resize, show, hide, and set the stacking order of the layer. These properties and methods are related to CSS style attributes and will be discussed in [Chapter 18](#). The most interesting thing about the Layer object is that it contains a Document object of its own: the content of a layer is treated as an entirely separate document from the document that contains the layer. This allows you to modify the content displayed by a layer by dynamically rewriting the content of the layer using the `document.write()` and `document.close()` methods. You can also dynamically load documents into a layer using Layer's `load()` method. Finally, note that layers may themselves contain layers, and you can refer to such nested layers with expressions like this:

```
// The second layer nested within the layer named "mylayer"
```

```
document.mylayer.document.layers[1];
```

17.5 Convenience Methods: The Traversal and Range APIs

So far in this chapter, we've discussed the core DOM API, which provides basic methods for document traversal and manipulation. The DOM standard also defines other optional API modules, the most important of which will be discussed in the chapters that follow. Two of the optional modules are essentially convenience APIs built on top of the core API. The Traversal API defines advanced techniques for traversing a document and filtering out nodes that are not of interest. The Range API defines methods for manipulating contiguous ranges of document content, even when that content does not begin or end at a node boundary. The Traversal and Range APIs are briefly introduced in the sections that follow. See the DOM reference section for complete documentation. The Range API is implemented by Netscape 6.1 (and partially implemented by Netscape 6), and the Traversal API is expected to be fully supported by Mozilla 1.0, which means that a future release of Netscape will support it. At the time of this writing, IE does not support either of these APIs.

17.5.1 The DOM Traversal API

At the beginning of this chapter, we saw techniques for traversing the document tree by recursively examining each node in turn. This is an important technique, but it is often overkill; we do not typically want to examine every node of a document. We instead might want to examine only the `` elements in a document, or to traverse only the subtrees of `<table>` elements. The Traversal API provides advanced techniques for this kind of selective document traversal. As noted previously, the Traversal API is optional and, at the time of this writing, is not implemented in major sixth-generation browsers. You can test whether it is supported by a DOM-compliant browser with the following:

```
document.implementation.hasFeature("Traversal", 2.0) // True if
supported
```

17.5.1.1 *NodeIterator and TreeWalker*

The Traversal API consists of two key objects, each of which provides a different filtered view of a document. The `NodeIterator` object provides a "flattened" sequential view of the nodes in a document and supports filtering. You could define a `NodeIterator` that filters out all document content except `` tags and presents those image elements to you as a list. The `nextNode()` and `previousNode()` methods of the `NodeIterator` object allow you to move forward and backward through the list. Note that `NodeIterator` allows you to traverse selected parts of a document without recursion; you can simply use a `NodeIterator` within a loop, calling `nextNode()` repeatedly until you find the node or nodes in which you are interested, or until it returns `null`, indicating that it has reached the end of the document.

The other key object in the Traversal API is `TreeWalker`. This object also provides a filtered view of a document and allows you to traverse the filtered document by calling `nextNode()` and `previousNode()`, but it does not flatten the document tree. `TreeWalker` retains the tree structure of the document (although this tree structure may be dramatically modified by node filtering) and allows you to navigate the tree with the `firstChild()`, `lastChild()`, `nextSibling()`, `previousSibling()`, and `parentNode()` methods. You would use a `TreeWalker` instead of a `NodeIterator` when you want to traverse the filtered tree yourself, instead of simply calling `nextNode()` to iterate through it, or when you want to perform a more sophisticated traversal, skipping, for example, some subtrees.

The `Document` object defines `createNodeIterator()` and `createTreeWalker()` methods for creating `NodeIterator` and `TreeWalker` objects. A practical way to check whether a browser supports the Traversal API is to test for the existence of these methods:

```
if (document.createNodeIterator && document.createTreeWalker) {  
    /* Safe to use Traversal API */  
}
```

Both `createNodeIterator()` and `createTreeWalker()` are passed the same four arguments and differ only in the type of object they return. The first argument is the node at which the traversal is to begin. This should be the `Document` object if you want to traverse or iterate through the entire document, or any other node if you want to traverse only a subtree of the document. The second argument is a number that indicates the types of nodes `NodeIterator` or `TreeWalker` should return. This argument is formed by taking the sum of one or more of the `SHOW_` constants defined by the `NodeFilter` object (discussed in the next section). The third argument to both methods is an optional function used to specify a more complex filter than simply including or rejecting nodes based on their type (again, see the next section). The final argument is a boolean value that specifies whether entity reference nodes in the document should be expanded during the traversal. This option can be useful when you're working with XML documents, but web programmers working with HTML documents can ignore it and pass `false`.

17.5.1.2 Filtering

One of the most important features of `NodeIterator` and `TreeWalker` is their selectivity, their ability to filter out nodes you don't care about. As described previously, you specify the nodes you are interested in with the second and (optionally) third arguments to `createNodeIterator()` and `createTreeWalker()`. These arguments specify two levels of filtering. The first level simply accepts or rejects nodes based on their type. The `NodeFilter` object defines a numeric constant for each type of node, and you specify the types of nodes you are interested in by adding together (or by using the `|` bitwise OR operator on) the appropriate constants.

For example, if you are interested in only the Element and Text nodes of a document, you can use the following expression as the second argument:

```
NodeFilter.SHOW_ELEMENT + NodeFilter.SHOW_TEXT
```

If you are interested in only Element nodes, use:

```
NodeFilter.SHOW_ELEMENT
```

If you are interested in all nodes or do not want to reject any nodes simply on the basis of their types, use the special constant:

```
NodeFilter.SHOW_ALL
```

And if you are interested in all types of nodes except for comments, use:

```
~NodeFilter.SHOW_COMMENT
```

(See [Chapter 5](#) if you've forgotten the meaning of the `~` operator.) Note that this first level of filtering applies to individual nodes but not to their children. If the second argument is `NodeFilter.SHOW_TEXT`, your `NodeIterator` or `TreeWalker` does not return element nodes to you, but it does not discard them entirely; it still traverses the subtree beneath the Element nodes to find the Text nodes you are interested in.

Any nodes that pass this type-based filtration may be put through a second level of filtering. This second filter is implemented by a function you define and can therefore perform arbitrarily complex filtering. If you do not need this kind of filtering, you can simply specify `null` as the value of the third argument to `createNodeIterator()` or `createTreeWalker()`. But if you do want this kind of filtering, you must pass a function as the third argument.

The function should expect a single node argument, and it should evaluate the node and return a value that indicates whether the node should be filtered out. There are three possible return values, defined by three `NodeFilter` constants. If your filter function returns `NodeFilter.FILTER_ACCEPT`, the node is returned by the `NodeIterator` or `TreeWalker`. If your function returns `NodeFilter.FILTER_SKIP`, the node is filtered out and is not returned by the `NodeIterator` or `TreeWalker`. The children of the node are still traversed, however. If you are working with a `TreeWalker`, your filter function may also return the value `NodeFilter.FILTER_REJECT`, which specifies that the node should not be returned and that it should not even be traversed.

[Example 17-10](#) demonstrates the creation and use of a NodeIterator and should clarify the previous discussion. Note, however, that at the time of this writing none of the major web browsers support the Traversal API, so this example is untested!

Example 17-10. Creating and using a NodeIterator

```
// Define a NodeFilter function to accept only <img> elements
function imgfilter(n) {
    if (n.tagName == 'IMG') return NodeFilter.FILTER_ACCEPT;
    else return NodeFilter.FILTER_SKIP;
}

// Create a NodeIterator to find <img> tags
var images = document.createNodeIterator(document, // Traverse entire
document
    /* Look only at Element nodes */      NodeFilter.SHOW_ELEMENT,
    /* Filter out all but <img> */        imgfilter,
    /* Unused in HTML documents */       false);

// Use the iterator to loop through all images and do something with
them
var image;
while((image = images.nextNode( )) != null) {
    image.style.visibility = "hidden"; // Process the image here
}
```

17.5.2 The DOM Range API

The DOM Range API consists of a single interface, Range. A Range object represents a contiguous range^[8] of document content, contained between a specified start position and a specified end position. Many applications that display text and documents allow the user to select a portion of the document by dragging with the mouse. Such a selected portion of a document is conceptually equivalent to a range.^[9] When a node of a document tree falls within a range, we often say that the node is "selected," even though the Range object may not have anything to do with a selection action initiated by the end user. When the start and end positions of a range are the same, we say that the range is "collapsed." In this case, the Range object represents a single position or insertion point within a document.

^[8] That is, a logically contiguous range. In bidirectional languages such as Arabic and Hebrew, a logically contiguous range of a document may be visually discontinuous when displayed.

^[9] Although web browsers typically allow the user to select document content, the current DOM Level 2 standard does not make the contents of those ranges available to JavaScript, so there is no standard way to obtain a Range object that corresponds to a user's desired selection.

The Range object provides methods for defining the start and end positions of a range, copying and deleting the contents of a range, and inserting nodes at the start position of a range. Support for the Range API is optional. At the time of this writing, it is supported by Netscape 6.1. IE 5 supports a proprietary API that is similar to, but not compatible with, the Range API. You can test for Range support with this code:


```
document.implementation.hasFeature("Range", "2.0"); // True if Range
is supported
```

17.5.2.1 Start and end positions

The start and end positions of a range are each specified by two values. The first value is a document node, typically a Document, Element, or Text object. The second value is a number that represents a position within that node. When the node is a document or element, the number represents a position between the children of the document or the element. An offset of 0, for example, represents the position immediately before the first child of the node. An offset of 1 represents the position after the first child and before the second. When the specified node is a Text node (or another text-based node type, such as Comment), the number represents a position between the characters of text. An offset of 0 specifies the position before the first character of text, an offset of 1 specifies the position between the first and second characters, and so on. With start and end positions specified in this way, a range represents all nodes and/or characters between the start and end positions. The real power of the Range interface is that the start and end positions may fall within different nodes of the document, and therefore a range may span multiple (and fractional) Element and Text nodes.

To demonstrate the action of the various range-manipulation methods, I'm going to adopt the notation used in the DOM specification for illustrating the document content represented by a range. Document contents are shown in the form of HTML source code, with the contents of a range in **bold**. For example, the following line represents a range that begins at position 0 within the `<body>` node and continues to position 8 within the Text node contained within the `<h1>` node:

```
<body><h1>Document Title</h1></body>
```

To create a Range object, call the `createRange()` method of the Document object:

```
var r = document.createRange( );
```

Newly created ranges have both start and end points initialized to position 0 within the Document object. Before you can do anything interesting with a range, you must set the start and end positions to specify the desired document range. There are several ways you can do this. The most general way is to call the `setStart()` and `setEnd()` methods to specify the start and end points. Each is passed a node and a position within the node.

A higher-level technique for setting a start and/or end position is to call `setStartBefore()`, `setStartAfter()`, `setEndBefore()`, or `setEndAfter()`. These methods each take a single node as their argument. They set the start or end position of the Range to the position before or after the specified node within the parent of that node.

Finally, if you want to define a Range that represents a single Node or subtree of a document, you can use the `selectNode()` or `selectNodeContent()` method. Both methods take a single node argument. `selectNode()` sets the start and end positions before and after the specified node within its parent, defining a range that includes the node and all of its children. `selectNodeContent()` sets the start of the range to the position before the first child of the node and sets the end of the range to the position after the last child of the node. The resulting range contains all the children of the specified node, but not the node itself.

17.5.2.2 Manipulating ranges

Once you've defined a range, there are a number of interesting things you can do with it. To delete the document content within a range, simply call the `deleteContents()` method of the Range object. When a range includes partially selected Text nodes, the deletion operation is a little tricky. Consider the following range:

```
<p>This is <i>only</i> a test
```

After a call to `deleteContents()`, the affected portion of the document looks like this:

```
<p>This<i>ly</i> a test
```

Even though the `<i>` element was included (partially) in the Range, that element remains (with modified content) in the document tree after the deletion.

If you want to remove the content of a range from a document but also want to save the extracted content (for reinsertion as part of a paste operation, perhaps), you should use `extractContents()` instead of `deleteContents()`. This method removes nodes from the document tree and inserts them into a DocumentFragment (introduced earlier in this chapter), which it returns. When a range includes a partially selected node, that node remains in the document tree and has its content modified as needed. A clone of the node (see `Node.cloneNode()`) is made (and modified) to insert into the DocumentFragment. Consider the previous example again. If `extractContents()` is called instead of `deleteContents()`, the effect on the document is the same as shown previously, and the returned DocumentFragment contains:

```
is <i>on</i>
```

`extractContents()` works when you want to perform the equivalent of a cut operation on the document. If instead you want to do a copy operation and extract content without deleting it from the document, use `cloneContents()` instead of `extractContents()`.^[10]

^[10] Implementing word processor-style cut, copy, and paste operations is actually more complex than this. Simple range operations on a complex document tree do not always produce the desired cut-and-paste behavior in the linear view of the document.

In addition to specifying the boundaries of text to be deleted or cloned, the start position of a range can be used to indicate an insertion point within a document. The `insertNode()` method of a range inserts the specified node (and all of its children) into the document at the start position of the range. If the specified node is already part of the document tree, it is moved from its current location and reinserted at the position specified by the range. If the specified node is a `DocumentFragment`, all the children of the node are inserted instead of the node itself.

Another useful method of the `Range` object is `surroundContents()`. This method reparents the contents of a range to the specified node and inserts that node into the document tree at the position of the range. For example, by passing a newly created `<i>` node to `surroundContents()`, you could transform this range:

```
This is only a test
```

into:

```
This is only a test
```

Note that because opening and closing tags must be properly nested in HTML files, `surroundContents()` cannot be used (and will throw an exception) for ranges that partially select any nodes other than `Text` nodes. The range used earlier to illustrate the `deleteContents()` method could not be used with `surroundContents()`, for example.

The `Range` object has various other features as well. You can compare the boundaries of two different ranges with `compareBoundaryPoints()`, clone a range with `cloneRange()`, and extract a plain-text copy of the content of a range (not including any markup) with `toString()`. The start and end positions of a range are accessible through the read-only properties `startContainer`, `startOffset`, `endContainer`, and `endOffset`. The start and end points of all valid ranges share a common ancestor somewhere in the document tree, even if it is the `Document` object at the root of the tree. You can find out what this common ancestor is with the `commonAncestorContainer` property of the range.

Chapter 18. Cascading Style Sheets and Dynamic HTML

Cascading Style Sheets (CSS) is a standard for specifying the visual presentation^[1] of HTML (or XML) documents. In theory, you use HTML markup to specify the structure of your document, resisting the temptation to use deprecated HTML tags such as ``

to specify how the document should look. Instead, you use CSS to define a style sheet that specifies how the structured elements of your document should be displayed. For example, you can use CSS to specify that the level-one headings defined by `<h1>` tags should be displayed in bold, sans-serif, centered, uppercase, 24-point letters.

^[1] And, in the CSS2 standard, also the aural presentation.

CSS is a technology intended for use by graphic designers or anyone concerned with the precise visual display of HTML documents. It is of interest to client-side JavaScript programmers because the document object model allows the styles that are applied to the individual elements of a document to be scripted. Used together, CSS and JavaScript enable a variety of visual effects loosely referred to as Dynamic HTML (DHTML).^[2]

^[2] Many advanced DHTML effects also involve the event-handling techniques we'll see in [Chapter 19](#).

The ability to script CSS styles allows you to dynamically change colors, fonts, and so on. More importantly, it allows you to set and change the position of elements and even to hide and show elements. This means that you can use DHTML techniques to create animated transitions where document content "slides in" from the right, for example, or an expanding and collapsing outline list in which the user can control the amount of information that is displayed.

This chapter begins with an overview of CSS style sheets and the use of CSS styles to specify the position and visibility of document elements. It then explains how CSS styles can be scripted using the API defined by the DOM Level 2 standard. Finally, it demonstrates the nonstandard, browser-specific APIs that can be used to achieve DHTML effects in Netscape 4 and Internet Explorer 4.

18.1 Styles and Style Sheets with CSS

Styles in CSS are specified as a semicolon-separated list of name/value attribute pairs, where each name and value are separated by colons. For example, the following style specifies bold, blue, underlined text:

```
font-weight: bold; color: blue; text-decoration: underline;
```

The CSS standard defines quite a few style attributes you can set. [Table 18-1](#) lists all the attributes except for those used only in audio style sheets. You are not expected to understand or be familiar with all these attributes, their values, or their meanings. As you become familiar with CSS and use it in your documents and scripts, however, you may find this table a convenient quick reference. For more complete documentation on CSS, consult *Cascading Style Sheets: The Definitive Guide*, by Eric Meyer (O'Reilly), or *Dynamic HTML: The Definitive Guide*, by Danny Goodman (O'Reilly). Or read the specification itself -- you can find it at <http://www.w3c.org/TR/REC-CSS2/>.

The second column of [Table 18-1](#) shows the allowed values for each style attribute. It uses the grammar used by the CSS specification. Items in *fixed-width font* are keywords and should appear exactly as shown. Items in *italics* specify a data type such as a string or a length. Note that the *length* type is a number followed by a units specification such as *px* (for pixels). See a CSS reference for details on the other types. Items that appear in *italic fixed-width font* represent the set of values allowed by some other CSS attribute. In addition to the values shown in the table, each style attribute may have the value "inherit", to specify that it should inherit the value from its parent.

Values separated by a | are alternatives; you must specify exactly one. Values separated by || are options; you must specify at least one, but you may specify more than one, and they can appear in any order. Square brackets [] are used for grouping values. An asterisk (*) specifies that the previous value or group may appear zero or more times, a plus sign (+) specifies that the previous value or group may appear one or more times, and a question mark (?) specifies that the previous item is optional and may appear zero or one time. Numbers within curly braces specify a number of repetitions. For example, {2} specifies that the previous item must be repeated twice, and {1,4} specifies that the previous item must appear at least once and no more than four times. (This repetition syntax may seem familiar: it is the same one used by JavaScript regular expressions, discussed in [Chapter 10](#).)

Table 18-1. CSS style attributes and their values

Name	Values
background	[<i>background-color</i> <i>background-image</i> <i>background-repeat</i> <i>background-attachment</i> <i>background-position</i>]
background-attachment	scroll fixed
background-color	<i>color</i> transparent
background-image	url(<i>url</i>) none
background-position	[[<i>percentage</i> <i>length</i>]{1,2} [[top center bottom] [left center right]]]
background-repeat	repeat repeat-x repeat-y no-repeat
border	[<i>border-width</i> <i>border-style</i> <i>color</i>]
border-collapse	collapse separate
border-color	<i>color</i> {1,4} transparent
border-spacing	<i>length length?</i>
border-style	[none hidden dotted dashed solid double groove

Table 18-1. CSS style attributes and their values

Name	Values
	ridge inset outset]{1,4}
border-top border-right border-bottom border-left	[<i>border-top-width</i> <i>border-top-style</i> <i>color</i>]
border-top-color border-right-color border-bottom-color border-left-color	<i>color</i>
border-top-style border-right-style border-bottom-style border-left-style	none hidden dotted dashed solid double groove ridge inset outset
border-top-width border-right-width border-bottom-width border-left-width	thin medium thick <i>length</i>
border-width	[thin medium thick <i>length</i>]{1,4}
bottom	<i>length</i> <i>percentage</i> auto
caption-side	top bottom left right
clear	none left right both
clip	[rect([<i>length</i> auto]{4})] auto
color	<i>color</i>
content	[<i>string</i> url(<i>url</i>) <i>counter</i> attr(<i>attribute-name</i>) open-quote close-quote no-open-quote no-close-quote]+
counter-increment	[<i>identifier integer?</i>]+ none
counter-reset	[<i>identifier integer?</i>]+ none
cursor	[[url(<i>url</i>) ,]* [auto crosshair default pointer move e-resize ne-resize nw-resize n-resize se-resize sw-resize s-resize w-resize text wait help]]
direction	ltr rtl
display	inline block list-item run-in compact marker table inline-table table-row-group table-header-group table-footer-group table-row table-column-group table-column table-cell table-caption none

Table 18-1. CSS style attributes and their values

Name	Values
empty-cells	show hide
float	left right none
font	[[<i>font-style</i> <i>font-variant</i> <i>font-weight</i>]? <i>font-size</i> [/ <i>line-height</i>]? <i>font-family</i>] caption icon menu message-box small-caption status-bar
font-family	[[<i>family-name</i> serif sans-serif monospace cursive fantasy],]+
font-size	xx-small x-small small medium large x-large xx-large smaller larger <i>length</i> <i>percentage</i>
font-size-adjust	<i>number</i> none
font-stretch	normal wider narrower ultra-condensed extra-condensed condensed semi-condensed semi-expanded expanded extra-expanded ultra-expanded
font-style	normal italic oblique
font-variant	normal small-caps
font-weight	normal bold bolder lighter 100 200 300 400 500 600 700 800 900
height	<i>length</i> <i>percentage</i> auto
left	<i>length</i> <i>percentage</i> auto
letter-spacing	normal <i>length</i>
line-height	normal <i>number</i> <i>length</i> <i>percentage</i>
list-style	[<i>list-style-type</i> <i>list-style-position</i> <i>list-style-image</i>]
list-style-image	url(<i>url</i>) none
list-style-position	inside outside
list-style-type	disc circle square decimal decimal-leading-zero lower-roman upper-roman lower-greek lower-alpha lower-latin upper-alpha upper-latin hebrew armenian georgian cjk-ideographic hiragana katakana hiragana-iroha katakana-iroha none
margin	[<i>length</i> <i>percentage</i> auto]{1,4}

Table 18-1. CSS style attributes and their values

Name	Values
margin-top margin-right margin-bottom margin-left	<i>length</i> <i>percentage</i> auto
marker-offset	<i>length</i> auto
marks	[<i>crop</i> <i>cross</i>] none
max-height	<i>length</i> <i>percentage</i> none
max-width	<i>length</i> <i>percentage</i> none
min-height	<i>length</i> <i>percentage</i>
min-width	<i>length</i> <i>percentage</i>
orphans	<i>integer</i>
outline	[<i>outline-color</i> <i>outline-style</i> <i>outline-width</i>]
outline-color	<i>color</i> invert
outline-style	none hidden dotted dashed solid double groove ridge inset outset
outline-width	thin medium thick <i>length</i>
overflow	visible hidden scroll auto
padding	[<i>length</i> <i>percentage</i>]{1,4}
padding-top padding-right padding-bottom padding-left	<i>length</i> <i>percentage</i>
page	<i>identifier</i> auto
page-break-after	auto always avoid left right
page-break-before	auto always avoid left right
page-break-inside	avoid auto
position	static relative absolute fixed
quotes	[<i>string string</i>]+ none
right	<i>length</i> <i>percentage</i> auto
size	<i>length</i> {1,2} auto portrait landscape
table-layout	auto fixed

Table 18-1. CSS style attributes and their values

Name	Values
text-align	left right center justify <i>string</i>
text-decoration	none [underline overline line-through blink]
text-indent	<i>length</i> <i>percentage</i>
text-shadow	none [<i>color</i> <i>length length length?</i> ,]* [<i>color</i> <i>length length length?</i>]
text-transform	capitalize uppercase lowercase none
top	<i>length</i> <i>percentage</i> auto
unicode-bidi	normal embed bidi-override
vertical-align	baseline sub super top text-top middle bottom text-bottom <i>percentage</i> <i>length</i>
visibility	visible hidden collapse
white-space	normal pre nowrap
widows	<i>integer</i>
width	<i>length</i> <i>percentage</i> auto
word-spacing	normal <i>length</i>
z-index	auto <i>integer</i>

The CSS standard allows certain style attributes that are commonly used together to be combined using special shortcut attributes. For example, the `font-family`, `font-size`, `font-style`, and `font-weight` attributes can all be set at once using a single `font` attribute:

```
font: bold italic 24pt helvetica;
```

In fact, some of the attributes listed in [Table 18-1](#) are themselves shortcuts. The `margin` and `padding` attributes are shortcuts for attributes that specify margins, padding, and borders for each of the individual sides of an element. Thus, instead of using the `margin` attribute, you can use `margin-left`, `margin-right`, `margin-top`, and `margin-bottom`, and similarly for `padding`.

18.1.1 Applying Style Rules to Document Elements

You can apply style attributes to the elements of a document in a number of ways. One way is to use them in the `style` attribute of an HTML tag. For example, to set the margins of an individual paragraph, you can use a tag like this:

```
<p style="margin-left: 1in; margin-right: 1in;">
```

One of the important goals of CSS is to separate document content and structure from document presentation. Specifying styles with the `style` attribute of individual HTML tags does not accomplish this (although it can be a useful technique for DHTML). To achieve the separation of structure from presentation, we use *style sheets*, which group all the style information into a single place. A CSS style sheet consists of a set of style rules. Each rule begins with a selector that specifies the document element or elements to which it applies, followed by a set of style attributes and their values within curly braces. The simplest kind of rule defines styles for one or more specific tag names. For example, the following rule sets the margins and background color for the `<body>` tag:

```
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff  
}
```

The following rule specifies that text within `<h1>` and `<h2>` headings should be centered:

```
h1, h2 { text-align: center; }
```

In the previous example, note the use of a comma to separate the tag names to which the styles are to apply. If the comma is omitted, the selector specifies a contextual rule that applies only when one tag is nested within another. For example, the following rules specify that `<blockquote>` tags are displayed in an italic font, but text inside an `<i>` tag inside a `<blockquote>` is displayed in plain, nonitalic text:

```
blockquote { font-style: italic; }  
blockquote i { font-style: normal; }
```

Another kind of style sheet rule uses a different selector to specify a *class* of elements to which its styles should be applied. The class of an element is defined by the `class` attribute of the HTML tag. For example, the following rule specifies that any tag with the attribute `class="attention"` should be displayed in bold:

```
.attention { font-weight: bold; }
```

Class selectors can be combined with tag name selectors. The following rule specifies that when a `<p>` tag has the `class="attention"` attribute, it should be displayed in red, in addition to being displayed in a bold font (as specified by the previous rule):

```
p.attention { color: red; }
```

Finally, style sheets can contain rules that apply only to individual elements that have a specified `id` attribute. The following rule specifies that the element with an `id` attribute equal to "p1" should not be shown:

```
#p1 { visibility: hidden; }
```

We've seen the `id` attribute before: it is used with the DOM function `getElementById()` to return individual elements of a document. As you might imagine, this kind of element-specific style sheet rule is useful when we want to manipulate the style of an individual element. Given the previous rule, for example, a script might switch the value of the `visibility` attribute from `hidden` to `visible`, causing the element to dynamically appear.

18.1.2 Associating Style Sheets with Documents

You can incorporate a style sheet into an HTML document by placing it between `<style>` and `</style>` tags within the `<head>` of the document, or you can store the style sheet in a file of its own and reference it from the HTML document using a `<link>` tag. You can also combine these two techniques by creating a document-specific style sheet between `<style>` tags that references or imports a document-independent style sheet using the special `@import` "at-rule." Consult a CSS reference for details on `@import`.

18.1.3 The Cascade

Recall that the `C` in CSS stands for "cascading." This term indicates that the style rules that apply to any given element in a document can come from a cascade of different sources. Each web browser typically has its own default styles for all HTML elements and may allow the user to override these defaults with a user style sheet. The author of a document can define style sheets within `<style>` tags or in external files that are linked in or imported into other style sheets. The author may also define inline styles for individual elements with the HTML `style` attribute.

The CSS specification includes a complete set of rules for determining which rules from the cascade take precedence over the other rules. Briefly, however, what you need to know is that the user style sheet overrides the default browser style sheet, author style sheets override the user style sheet, and inline styles override everything. The exception to this general rule is that user style attributes whose values include the `!important`

modifier override author styles. Within a style sheet, if more than one rule applies to an element, styles defined by the most specific rule override conflicting styles defined by less specific rules. Rules that specify an element `id` are the most specific. Rules that specify a `class` are next. Rules that specify only tag names are the least specific, but rules that specify multiple nested tag names are more specific than rules that specify only a single tag name.

18.1.4 Versions of CSS

At the time of this writing, there are two versions of the CSS standard. CSS1 was adopted in December, 1996 and defines attributes for specifying colors, fonts, margins, borders, and other basic styles. Netscape 4 and Internet Explorer 4 both implement at least partial support for CSS1. The second edition of the standard, CSS2, was adopted in May, 1998; it defines a number of more advanced features, most notably support for absolute positioning of elements. The advanced features of CSS2 are supported only in sixth-generation browsers. Fortunately, however, the crucial positioning features of CSS2 began the standardization process as part of a separate CSS-Positioning (CSS-P) effort, and therefore some of these DHTML-enabling features are available in fourth-generation browsers. Work continues on a third edition of the CSS standard. You can find the CSS specifications and working drafts at <http://www.w3.org/Style/CSS/>.

18.1.5 CSS Example

[Example 18-1](#) is an HTML file that defines and uses a style sheet. It demonstrates the previously described tag name, class, and ID-based style rules, and it also has an example of an inline style defined with the `style` attribute. Remember that this example is meant only as an overview of CSS syntax and capabilities. Full coverage of CSS is beyond the scope of this book.

Example 18-1. Defining and using Cascading Style Sheets

```
<head>
<style type="text/css">
/* Specify that headings display in blue italic text. */
h1, h2 { color: blue; font-style: italic }

/*
 * Any element of class="WARNING" displays in big bold text with large
 margins
 * and a yellow background with a fat red border.
 */
.WARNING {
    font-weight: bold;
    font-size: 150%;
    margin: 0 1in 0 1in; /* top right bottom left */
    background-color: yellow;
    border: solid red 8px;
    padding: 10px;      /* 10 pixels on all 4 sides */
}
```

```

/*
 * Text within an h1 or h2 heading within an element with
class="WARNING"
 * should be centered, in addition to appearing in blue italics.
 */
.WARNING h1, .WARNING h2 { text-align: center }

/* The single element with id="P23" displays in centered uppercase. */
#P23 {
    text-align: center;
    text-transform: uppercase;
}
</style>
</head>
<body>
<h1>Cascading Style Sheets Demo</h1>

<div class="WARNING">
<h2>Warning</h2>
This is a warning!
Notice how it grabs your attention with its bold text and bright
colors.
Also notice that the heading is centered and in blue italics.
</div>

<p id="P23">
This paragraph is centered<br>
and appears in uppercase letters.<br>
<span style="text-transform: none">
Here we explicitly use an inline style to override the uppercase
letters.
</span>
</p>
</body>

```

18.2 Element Positioning with CSS

For DHTML content developers, the most important feature of CSS is the ability to use ordinary CSS style attributes to specify the visibility, size, and precise position of individual elements of a document. In order to do DHTML programming, it is important to understand how these style attributes work. They are summarized in [Table 18-2](#) and documented in more detail in the sections that follow.

Table 18-2. CSS positioning and visibility attributes	
Attribute(s)	Description
position	Specifies the type of positioning applied to an element
top, left	Specifies the position of the top and left edges of an element
bottom,	Specifies the position of the bottom and right edges of an element

Table 18-2. CSS positioning and visibility attributes

Attribute(s)	Description
<code>right</code>	
<code>width, height</code>	Specifies the size of an element
<code>z-index</code>	Specifies the "stacking order" of an element relative to any overlapping elements; defines a third dimension of element positioning
<code>display</code>	Specifies how and whether an element is displayed
<code>visibility</code>	Specifies whether an element is visible
<code>clip</code>	Defines a "clipping region" for an element; only portions of the element within this region are displayed
<code>overflow</code>	Specifies what to do if an element is bigger than the space allotted for it

18.2.1 The Key to DHTML: The position Attribute

The CSS `position` attribute specifies the type of positioning applied to an element. The four possible values for this attribute are:

`static`

This is the default value and specifies that the element is positioned according to the normal flow of document content (for most Western languages, this is left to right and top to bottom.) Statically positioned elements are not DHTML elements and cannot be positioned with the `top`, `left`, and other attributes. To use DHTML positioning techniques with a document element, you must first set its `position` attribute to one of the other three values.

`absolute`

This value allows you to specify the position of an element relative to its containing element. Absolutely positioned elements are positioned independently of all other elements and are not part of the flow of statically positioned elements. An absolutely positioned element is positioned either relative to the `<body>` of the document or, if it is nested within another absolutely positioned element, relative to that element. This is the most commonly used positioning type for DHTML.

`fixed`

This value allows you to specify an element's position with respect to the browser window. Elements with `fixed` positioning do not scroll with the rest of the document and thus can be used to achieve frame-like effects. Like absolutely positioned elements, fixed-position elements are independent of all others and are not part of the document flow. Fixed positioning is a CSS2 feature and is not supported by fourth-generation browsers. (It is supported in Netscape 6 and IE 5 for the Macintosh, but it is not supported by IE 5 or IE 6 for Windows).

`relative`

When the `position` attribute is set to `relative`, an element is laid out according to the normal flow, and its position is then adjusted relative to its position in the normal flow. The space allocated for the element in the normal document flow remains allocated for it, and the elements on either side of it do not close up to fill in that space, nor are they "pushed away" from the new position of the element. Relative positioning can be useful for some static graphic design purposes, but it is not commonly used for DHTML effects.

18.2.2 Specifying the Position and Size of Elements

Once you have set the `position` attribute of an element to something other than `static`, you can specify the position of that element with some combination of the `left`, `top`, `right`, and `bottom` attributes. The most common positioning technique is to specify the `left` and `top` attributes, which specify the distance from the left edge of the containing element (usually the document itself) to the left edge of the element, and the distance from the top edge of the container to the top edge of the element. For example, to place an element 100 pixels from the left and 100 pixels from the top of the document, you can specify CSS styles in a `style` attribute as follows:

```
<div style="position: absolute; left: 100px; top: 100px;">
```

The containing element relative to which a dynamic element is positioned is not necessarily the same as the containing element within which the element is defined in the document source. Since dynamic elements are not part of normal element flow, their positions are not specified relative to the static container element within which they are defined. Most dynamic elements are positioned relative to the document (the `<body>` tag) itself. The exception is dynamic elements that are defined within other dynamic elements. In this case, the nested dynamic element is positioned relative to its nearest dynamic ancestor.

Although it is most common to specify the position of the upper-left corner of an element with `left` and `top`, you can also use `right` and `bottom` to specify the position of the bottom and right edges of an element relative to the bottom and right edges of the containing element. For example, to position an element so that its bottom-right corner is

at the bottom-right of the document (assuming it is not nested within another dynamic element), use the following styles:

```
position: absolute; right: 0px; bottom: 0px;
```

To position an element so that its top edge is 10 pixels from the top of the window and its right edge is 10 pixels from the right of the window, you can use these styles:

```
position: fixed; right: 10px; top: 10px;
```

Note that the `right` and `bottom` attributes are newer additions to the CSS standard and are not supported by fourth-generation browsers, as `top` and `left` are.

In addition to the position of elements, CSS allows you to specify their size. This is most commonly done by providing values for the `width` and `height` style attributes. For example, the following HTML creates an absolutely positioned element with no content. Its `width`, `height`, and `background-color` attributes make it appear as a small blue square:

```
<div style="position: absolute; left: 10px; right: 10px;
           width: 10px; height: 10px; background-color: blue">
</div>
```

Another way to specify the width of an element is to specify a value for both the `left` and `right` attributes. Similarly, you can specify the height of an element by specifying both `top` and `bottom`. If you specify a value for `left`, `right`, and `width`, however, the `width` attribute overrides the `right` attribute; if the height of an element is over-constrained, `height` takes priority over `bottom`.

Bear in mind that it is not necessary to specify the size of every dynamic element. Some elements, such as images, have an intrinsic size. Furthermore, for dynamic elements that contain text or other flowed content, it is often sufficient to specify the desired width of the element and allow the height to be determined automatically by the layout of the element's content.

In the previous positioning examples, values for the position and size attributes were specified with the suffix "px". This stands for pixels. The CSS standard allows measurements to be done in a number of other units, including inches ("in"), centimeters ("cm"), points ("pt"), and ems ("em" -- a measure of the line height for the current font). Pixel units are most commonly used with DHTML programming. Note that the CSS standard requires a unit to be specified. Some browsers may assume pixels if you omit the unit specification, but you should not rely on this behavior.

Instead of specifying absolute positions and sizes using the units shown above, CSS also allows you to specify the position and size of an element as a percentage of the size of the containing element. For example, the following HTML creates an empty element with a black border that is half as wide and half as high as the containing element (or the browser window) and centered within that element:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%;  
height: 50%;  
border: 2px solid black">  
</div>
```

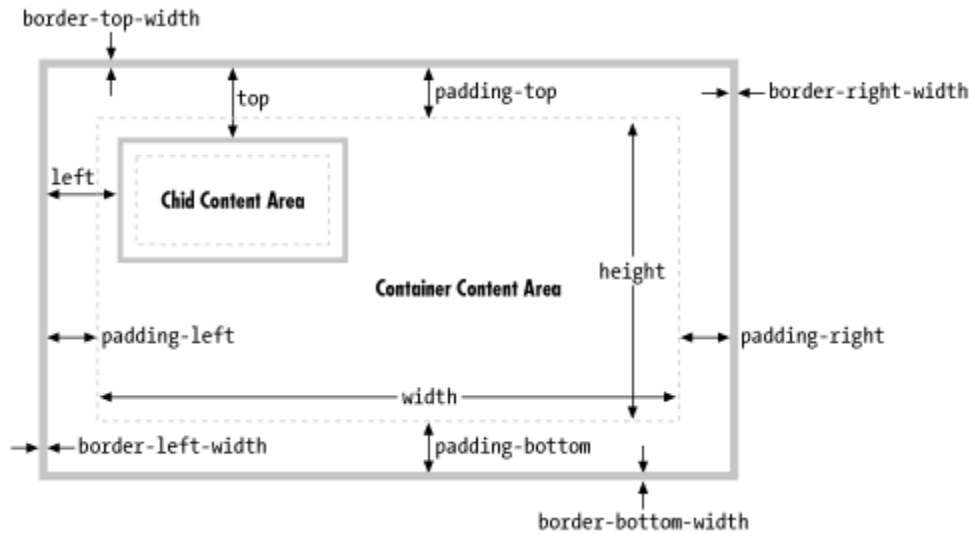
18.2.2.1 Element size and position details

It is important to understand some details about how the `left`, `right`, `width`, `top`, `bottom`, and `height` attributes work. First, `width` and `height` specify the size of an element's content area only; they do not include any additional space required for the element's padding, border, or margins. To determine the full onscreen size of an element with a border, you must add the left and right padding and left and right border widths to the element width, and you must add the top and bottom padding and top and bottom border widths to the element's height.

Since `width` and `height` specify the element content area only, you might think that `left` and `top` (and `right` and `bottom`) would be measured relative to the content area of the containing element. In fact, the CSS standard specifies that these values are measured relative to the outside edge of the containing element's padding (which is the same as the inside edge of the element's border).

Let's consider an example to make this clearer. Suppose you've created a dynamically positioned container element that has 10 pixels of padding all the way around its content area and a 5 pixel border all the way around the padding. Now suppose you dynamically position a child element inside this container. If you set the `left` attribute of the child to "0 px", you'll discover that the child is positioned with its left edge right up against the inner edge of the container's border. With this setting, the child overlaps the container's padding, which presumably was supposed to remain empty (since that is the purpose of padding). If you want to position the child element in the upper left corner of the container's content area, you should set both the `left` and `top` attributes to "10px". [Figure 18-1](#) helps to clarify this.

Figure 18-1. Dynamically positioned container and child elements with some CSS attributes



Now that you understand that `width` and `height` specify the size of an element's content area only and that the `left`, `top`, `right`, and `bottom` attributes are measured relative to the containing element's padding, there is one more detail you must be aware of: Internet Explorer Versions 4 through 5.5 for Windows (but not IE 5 for the Mac) implement the `width` and `height` attributes incorrectly and include an element's border and padding (but not its margins). For example, if you set the width of an element to 100 pixels and place a 10-pixel margin and a 5-pixel border on the left and right, the content area of the element ends up being only 70 pixels wide in these buggy versions of Internet Explorer.

In IE 6, the CSS position and size attributes work correctly when the browser is in standards mode and incorrectly (but compatibly with earlier versions) when the browser is in compatibility mode. Standards mode, and hence correct implementation of the CSS "box model," is triggered by the presence of a `<!DOCTYPE>` tag at the start of the document, declaring that the document adheres to the HTML 4.0 (or later) standard or some version of the XHTML standards. For example, any of the following three HTML document type declarations cause IE 6 to display documents in standards mode:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Strict//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

Netscape 6 and the Mozilla browser handle the `width` and `height` attributes correctly. But these browsers also have standards and compatibility modes, just as IE does. The absence of a `<!DOCTYPE>` declaration puts the Netscape browser in quirks mode, in which it mimics certain (relatively minor) nonstandard layout behaviors of Netscape 4. The presence of `<!DOCTYPE>` causes the browser to break compatibility with Netscape 4 and correctly implement the standards.

18.2.3 The Third Dimension: z-index

We've seen that the `left`, `top`, `right`, and `bottom` attributes can be used to specify the X and Y coordinates of an element within the two-dimensional plane of the containing element. The `z-index` attribute defines a kind of third dimension: it allows you to specify the stacking order of elements and indicate which of two or more overlapping elements is drawn on top of the others. The `z-index` attribute is an integer. The default value is zero, but you may specify positive or negative values (although fourth-generation browsers may not support negative `z-index` values). When two or more elements overlap, they are drawn in order from lowest to highest `z-index`; the element with the highest `z-index` appears on top of all the others. If overlapping elements have the same `z-index`, they are drawn in the order in which they appear in the document, so the last overlapping element appears on top.

Note that `z-index` stacking applies only to sibling elements (i.e., elements that are children of the same container). If two elements that are not siblings overlap, setting their individual `z-index` attributes does not allow you to specify which one is on top. Instead, you must specify the `z-index` attribute for the two sibling containers of the two overlapping elements.

Nonpositioned elements are always laid out in a way that prevents overlaps, so the `z-index` attribute does not apply to them. Nevertheless, they have a default `z-index` of zero, which means that positioned elements with a positive `z-index` appear on top of the normal document flow, and positioned elements with a negative `z-index` appear beneath the normal document flow.

Note, finally, that some browsers do not honor the `z-index` attribute when it is applied to `<iframe>` tags, and you may find that inline frames float on top of other elements, regardless of the specified stacking order. You may have the same problem with other "windowed" elements such as `<select>` drop-down menus. Fourth-generation browsers may display all form-control elements on top of absolutely positioned elements, regardless of `z-index` settings.

18.2.4 Element Display and Visibility

There are two CSS attributes you can use to affect the visibility of a document element: `visibility` and `display`. The `visibility` attribute is simple: when the attribute is set to the value `hidden`, the element is not shown; when it is set to the value `visible`, the element is shown. The `display` attribute is more general and is used to specify the type of display an item receives. It specifies whether an element is a block element, an inline element, a list item, and so on. When `display` is set to `none`, however, the affected element is not displayed, or even laid out, at all.

The difference between the `visibility` and `display` style attributes has to do with their effect on elements that are not dynamically positioned. For an element that appears in the

normal layout flow (with the `position` attribute set to `static` or `relative`), setting `visibility` to `none` makes the element invisible but reserves space for it in the document layout. Such an element can be repeatedly hidden and shown without changing the document layout. If an element's `display` attribute is set to `none`, however, no space is allocated for it in the document layout; elements on either side of it close up as if it were not there. (`visibility` and `display` have equivalent effects when used with absolute- or fixed-position elements, since these elements are never part of the document layout anyway.) You'll typically use the `visibility` attribute when you are working with dynamically positioned elements. The `display` attribute is useful when creating things like expanding and collapsing outlines.

Note that it doesn't make much sense to use `visibility` or `display` to make an element invisible unless you are going to use JavaScript to dynamically set these attributes and make the element visible at some point!^[3] You'll see how you can do this later in the chapter.

^[3] There is an exception: if you are creating a document that depends on CSS, you can warn users of browsers that do not support CSS with code like this:

18.2.5 Partial Visibility: overflow and clip

The `visibility` attribute allows you to completely hide a document element. The `overflow` and `clip` attributes allow you to display only part of an element. The `overflow` attribute specifies what happens when the content of an element exceeds the size specified (with the `width` and `height` style attributes, for example) for the element. The allowed values and their meanings for this attribute are as follows:

`visible`

Content may overflow and be drawn outside of the element's box if necessary. This is the default.

`hidden`

Content that overflows is clipped and hidden so that no content is ever drawn outside the region defined by the size and positioning attributes.

`scroll`

The element's box has permanent horizontal and vertical scrollbars. If the content exceeds the size of the box, the scrollbars allow the user to scroll to view the extra content. This value is honored only when the document is displayed on a computer screen; when the document is printed on paper, for example, scrollbars obviously do not make sense.

auto

Scrollbars are displayed only when content exceeds the element's size, rather than being permanently displayed.

While the `overflow` property allows you to specify what happens when an element's content is bigger than the element's box, the `clip` property allows you to specify exactly which portion of an element should be displayed, whether or not the element overflows. This attribute is especially useful for scripted DHTML effects in which an element is progressively displayed or uncovered.

The value of the `clip` property specifies the clipping region for the element. In CSS2 clipping regions are rectangular, but the syntax of the `clip` attribute leaves open the possibility that future versions of the standard will support clipping shapes other than rectangles. The syntax of the `clip` attribute is:

```
rect(top right bottom left)
```

The `top`, `right`, `bottom`, and `left` values specify the boundaries of the clipping rectangle relative to the upper-left corner of the element's box.^[4] For example, to display only a 100 x 100-pixel portion of an element, you can give that element this `style` attribute:

^[4] As the CSS2 specification was originally written, these four values specified the offset of the edges of the clipping region from each of the corresponding edges of the element's box. All major browser implementations got it wrong, however, and interpreted the `right` and `bottom` values as offsets from the left and top edges. Because the implementations consistently disagree with the specification, the specification is being modified to match the implementations.

```
style="clip: rect(0px 100px 100px 0px);"
```

Note that the four values within the parentheses are length values and must include a unit specification, such as "px" for pixels. Percentages are not allowed. Values may be negative to specify that the clipping region extends beyond the box specified for the element. You may also use the keyword `auto` for any of the four values to specify that that edge of the clipping region is the same as the corresponding edge of the element's box. For example, you can display just the leftmost 100 pixels of an element with this `style` attribute:

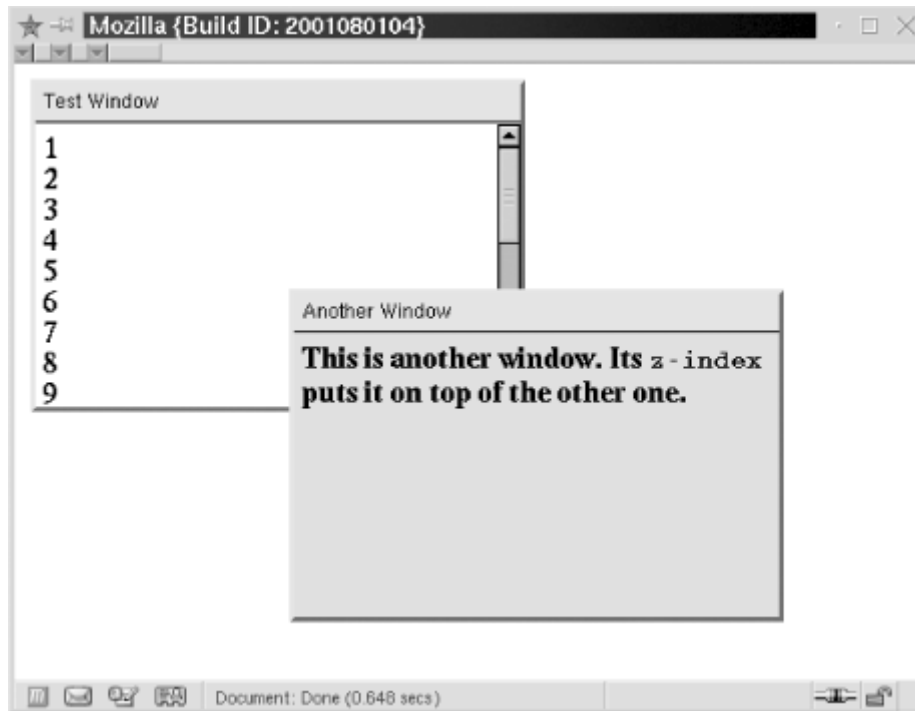
```
style="clip: rect(auto 100px auto auto);"
```

Note that there are no commas between the values, and the edges of the clipping region are specified in clockwise order from the top edge.

18.2.6 CSS Positioning Example

[Example 18-2](#) is a nontrivial example using CSS style sheets and CSS positioning attributes. When this HTML document is displayed in a CSS-compliant browser, it creates the visual effect of "subwindows" within the browser window. [Figure 18-2](#) shows the effect created by the code in [Example 18-2](#). Although the listing contains no JavaScript code, it is a useful demonstration of the powerful effects that can be achieved with CSS in general and the CSS positioning attributes in particular.

Figure 18-2. Windows created with CSS



Example 18-2. Displaying windows with CSS

```
<head>
<style type="text/css">
/**
 * This is a CSS style sheet that defines three style rules that we use
 * in the body of the document to create a "window" visual effect.
 * The rules use positioning attributes to set the overall size of the
window
 * and the position of its components. Changing the size of the window
 * requires careful changes to positioning attributes in all three
rules.
**/
div.window { /* Specifies size and border of the window */
  position: absolute; /* The position is specified
elsewhere */
  width: 300px; height: 200px; /* Window size, not including
borders */
  border: outset gray 3px; /* Note 3D "outset" border effect */
```

```

}

div.titlebar { /* Specifies position, size, and style of the titlebar
*/
    position: absolute; /* It's a positioned element */
    top: 0px; height: 18px; /* Titlebar is 18px + padding and borders
*/
    width: 290px; /* 290 + 5px padding on left and right =
300 */
    background-color: ActiveCaption; /* Use system titlebar color */
    border-bottom: groove black 2px; /* Titlebar has border on bottom
only */
    padding: 3px 5px 2px 5px; /* Values clockwise: top, right, bottom,
left */
    font: caption; /* Use system font for titlebar */
}

div.content { /* Specifies size, position and scrolling for window
content */
    position: absolute; /* It's a positioned element */
    top: 25px; /* 18px title+2px border+3px+2px
padding */
    height: 165px; /* 200px total - 25px titlebar - 10px
padding */
    width: 290px; /* 300px width - 10px of padding */
    padding: 5px; /* Allow space on all four sides */
    overflow: auto; /* Give us scrollbars if we need them
*/
    background-color: #ffffff; /* White background by default */
}
</style>
</head>

<body>
<!-- Here is how we define a window: a "window" div with a titlebar and
-->
<!-- content div nested between them. Note how position is specified
with -->
<!-- a style attribute that augments the styles from the style sheet. -
->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Test Window</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br> <!-- Lots of lines
to -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br> <!-- demonstrate
scrolling -->
</div>
</div>

<!-- Here's another window with different position, color, and font
weight -->
<div class="window" style="left: 170px; top: 140px; z-index: 20;">
<div class="titlebar">Another Window</div>
<div class="content" style="background-color:#d0d0d0; font-
weight:bold;">

```

```
This is another window. Its <tt>z-index</tt> puts it on top of the
other one.
</div>
</div>
</body>
```

The major shortcoming of this example is that the style sheet specifies a fixed size for all windows. Because the titlebar and content portions of the window must be precisely positioned within the overall window, changing the size of a window requires changing the value of various positioning attributes in all three rules defined by the style sheet. This is difficult to do in a static HTML document, but it would not be so difficult if we could use a script to set all of the necessary attributes. We'll explore this topic in the next section.

18.3 Scripting Styles

The crux of DHTML is the ability to use JavaScript to dynamically change the style attributes applied to individual elements within a document. The DOM Level 2 standard defines an API that makes this quite easy to do. In [Chapter 17](#), we saw how to use the DOM API to obtain references to document elements either by tag name or ID or by recursively traversing the entire document. Once you've obtained a reference to the element whose styles you want to manipulate, you use the element's `style` property to obtain a `CSS2Properties` object for that document element. This JavaScript object has JavaScript properties corresponding to each of the CSS1 and CSS2 style attributes. Setting these properties has the same effect as setting the corresponding styles in a `style` attribute on the element. Reading these properties returns the CSS attribute value, if any, that was set in the `style` attribute of the element. It is important to understand that the `CSS2Properties` object you obtain with the `style` property of an element specifies only the inline styles of the element. You cannot use the properties of the `CSS2Properties` object to obtain information about the style-sheet styles that apply to the element. By setting properties on this object, you are defining inline styles that effectively override style-sheet styles.

Consider the following script, for example. It finds all `` elements in the document and loops through them looking for ones that appear (based on their size) to be banner advertisements. When it finds an ad, it uses the `style.visibility` property to set the CSS `visibility` attribute to `hidden`, making the ad invisible:

```
var imgs = document.getElementsByTagName("img"); // Find all images
for(var i = 0; i < imgs.length; i++) { // Loop through them
    var img=imgs[i];
    if (img.width == 468 && img.height == 60) // If it's a 468x60
    banner...
        img.style.visibility = "hidden"; // hide it!
}
```


I've transformed this simple script into a "bookmarklet" by converting it to a `javascript:` URL and bookmarking it in my browser. I take subversive pleasure in using the bookmarklet to immediately hide distracting animated ads that won't stop animating. Here's a version of the script suitable for bookmarking:

```
javascript:a=document.getElementsByTagName("img");for(n=0;n<a.length;n+
+){
i=a[n];if(i.width==468&&i.height==60)i.style.visibility="hidden";}void
0;
```

The bookmarklet is written with very compact code and is intended to be formatted on a single line. The `javascript:` at the beginning of this bookmarklet identifies it as a URL whose body is a string of executable content. The `void 0` statement at the end causes the code to return an undefined value, which means that the browser continues to display the current web page (minus its banner ads, of course!). Without the `void 0`, the browser would overwrite the current web page with the return value of the last JavaScript statement executed.

18.3.1 Naming Conventions: CSS Attributes in JavaScript

Many CSS style attributes, such as `font-family`, contain hyphens in their names. In JavaScript, a hyphen is interpreted as a minus sign, so it is not possible to write an expression like:

```
element.style.font-family = "sans-serif";
```

Therefore, the names of the properties of the `CSS2Properties` object are slightly different from the names of actual CSS attributes. If a CSS attribute name contains one or more hyphens, the `CSS2Properties` property name is formed by removing the hyphens and capitalizing the letter immediately following each hyphen. Thus, the `border-left-width` attribute is accessed through the `borderLeftWidth` property, and you can access the `font-family` attribute with code like this:

```
element.style.fontFamily = "sans-serif";
```

There is one other naming difference between CSS attributes and the JavaScript properties of `CSS2Properties`. The word "float" is a keyword in Java and other languages, and although it is not currently used in JavaScript, it is reserved for possible future use. Therefore, the `CSS2Properties` object cannot have a property named `float` to correspond to the CSS `float` attribute. The solution to this problem is to prefix the `float` attribute with the string "css" to form the property name `cssFloat`. Thus, to set or query the value of the `float` attribute of an element, use the `cssFloat` property of the `CSS2Properties` object.

18.3.2 Working with Style Properties

When working with the style properties of the `CSS2Properties` object, remember that all values must be specified as strings. In a style sheet or `style` attribute, you can write:

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

To accomplish the same thing for an element `e` with JavaScript, you have to quote all of the values:

```
e.style.position = "absolute";  
e.style.fontFamily = "sans-serif";  
e.style.backgroundColor = "#ffffff";
```

Note that the semicolons go outside the strings. These are just normal JavaScript semicolons; the semicolons you use in CSS style sheets are not required as part of the string values you set with JavaScript.

Furthermore, remember that all the positioning properties require units. Thus, it is not correct to set the `left` property like this:

```
e.style.left = 300;    // Incorrect: this is a number, not a string  
e.style.left = "300"; // Incorrect: the units are missing
```

Units are required when setting style properties in JavaScript, just as they are when setting style attributes in style sheets. The correct way to set the value of the `left` property of an element `e` to 300 pixels is:

```
e.style.left = "300px";
```

If you want to set the `left` property to a computed value, be sure to append the units at the end of the computation:

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

As a side effect of appending the units, the addition of the unit string converts the computed value from a number to a string.

You can also use the `CSS2Properties` object to query the values of the CSS attributes that were explicitly set in the `style` attribute of an element or to read any inline style values previously set by JavaScript code. Once again, however, you must remember that the values returned by these properties are strings, not numbers, so the following code (which

assumes that the element `e` has its margins specified with inline styles) does not do what you might expect it to:

```
var totalMarginWidth = e.style.marginLeft + e.style.marginRight;
```

Instead, you should use code like this:

```
var totalMarginWidth = parseInt(e.style.marginLeft) +  
parseInt(e.style.marginRight);
```

This expression simply discards the unit specifications returned at the ends of both strings. It assumes that both the `marginLeft` and `marginRight` properties were specified using the same units. If you exclusively use pixel units in your inline styles, you can usually get away with discarding the units like this.

Recall that some CSS attributes, such as `margin`, are shortcuts for other properties, such as `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`. The `CSS2Properties` object has properties that correspond to these shortcut attributes. For example, you might set the `margin` property like this:

```
e.style.margin = topMargin + "px " + rightMargin + "px " +  
                bottomMargin + "px " + leftMargin + "px";
```

Arguably, it is easier to set the four margin properties individually:

```
e.style.marginTop = topMargin + "px";  
e.style.marginRight = rightMargin + "px";  
e.style.marginBottom = bottomMargin + "px";  
e.style.marginLeft = leftMargin + "px";
```

You can also query the values of shortcut properties, but this is rarely worthwhile, because typically you must then parse the returned value to break it up into its component parts. This is usually difficult to do, and it is much simpler to query the component properties individually.

Finally, let me emphasize again that when you obtain a `CSS2Properties` object from the `style` property of an `HTMLElement`, the properties of this object represent the values of inline style attributes for the element. In other words, setting one of these properties is like setting a CSS attribute in the `style` attribute of the element: it affects only that one element, and it takes precedence over conflicting style settings from all other sources in the CSS cascade. This precise control over individual elements is exactly what we want when using JavaScript to create DHTML effects.

When you read the values of these `CSS2Properties` properties, however, they return meaningful values only if they've previously been set by your JavaScript code or if the HTML element with which you are working has an inline `style` attribute that sets the desired property. For example, your document may include a style sheet that sets the left margin for all paragraphs to 30 pixels, but if you read the `leftMargin` property of one of your paragraph elements, you'll get the empty string unless that paragraph has a `style` attribute that overrides the style sheet setting. Thus, although the `CSS2Properties` object is useful for setting styles that override any other styles, it does not provide a way to query the CSS cascade and determine the complete set of styles that apply to a given element. Later in this chapter we will briefly consider the `getComputedStyle()` method, which does provide this ability.

18.3.3 Example: Dynamic Bar Charts

When adding graphs and charts to your HTML documents, you typically implement them as static, inline images. Because the CSS layout model is heavily based on rectangular boxes, however, it is possible to dynamically create bar charts using JavaScript, HTML, and CSS. [Example 18-3](#) shows how this can be done. This example defines a function `makeBarChart()` that makes it simple to insert bar charts into your HTML documents.

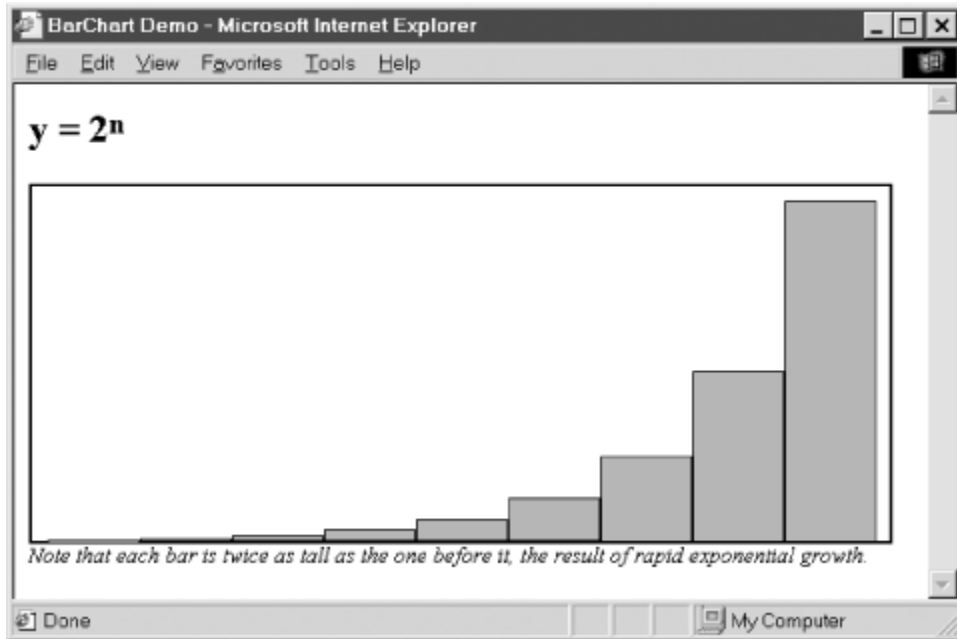
The code for [Example 18-3](#) uses the techniques shown in [Chapter 17](#) to create new `<div>` elements and add them to the document and the techniques discussed in this chapter to set style properties on the elements it creates. No text or other content is involved; the bar chart is just a bunch of rectangles carefully sized and positioned within another rectangle. CSS `border` and `background-color` attributes are used to make the rectangles visible.

The example includes some simple math to compute the height in pixels of each bar based on the values of the data to be charted. The JavaScript code that sets the position and size of the chart and its bars also includes some simple arithmetic to account for the presence of borders and padding. With the techniques shown in this example, you should be able to modify [Example 18-2](#) to include a JavaScript function that dynamically creates windows of any specified size.

[Figure 18-3](#) shows a bar chart created using the `makeBarChart()` function as follows:

```
<html>
<head>
<title>BarChart Demo</title>
<script src="BarChart.js"></script>
</head>
<body>
<h1>y = 2<sup>n</sup></h1>
<script>makeBarChart([2,4,8,16,32,64,128,256,512], 600, 250,
"red");</script>
<i>Note that each bar is twice as tall as the one before it,
the result of rapid exponential growth.</i>
</body>
</html>
```

Figure 18-3. A dynamically created bar chart



Example 18-3. Dynamically creating bar charts

```
/**
 * BarChart.js:
 * This file defines makeBarChart( ), a function that creates a bar
 * chart to
 * display the numbers from the data[] array. The chart is a block
 * element
 * inserted at the current end of the document. The overall size of the
 * chart
 * is specified by the optional width and height arguments, which
 * include the
 * space required for the chart borders and internal padding. The
 * optional
 * barcolor argument specifies the color of the bars. The function
 * returns the
 * chart element it creates, so the caller can further manipulate it by
 * setting a margin size, for example.
 *
 * Import this function into an HTML file with code like this:
 * <script src="BarChart.js"></script>
 * Use this function in an HTML file with code like this:
 * <script>makeBarChart([1,4,9,16,25], 300, 150, "yellow");</script>
 */
function makeBarChart(data, width, height, barcolor) {
  // Provide default values for the optional arguments
  if (!width) width = 500;
  if (!height) height = 350;
  if (!barcolor) barcolor = "blue";
```

```

    // The width and height arguments specify the overall size of the
    // generated chart. We have to subtract the border and padding
    // sizes to get the size of the element we create.
    width -= 24; // Subtract 10px padding and 2px left and right
border
    height -= 14; // Subtract 10px top padding and 2px top and bottom
border

    // Now create an element to hold the chart. Note that we make the
    chart
    // relatively positioned so that it can have absolutely positioned
    children,
    // but it still appears in the normal element flow.
    var chart = document.createElement("DIV");
    chart.style.position = "relative"; // Set relative
positioning
    chart.style.width = width + "px"; // Set the chart width
    chart.style.height = height + "px"; // Set the chart height
    chart.style.border = "solid black 2px"; // Give it a border
    chart.style.paddingLeft = "10px"; // Add padding on the
left,
    chart.style.paddingRight = "10px"; // on the right,
    chart.style.paddingTop = "10px"; // and on the top,
    chart.style.paddingBottom = "0px"; // but not on the bottom
    chart.style.backgroundColor = "white"; // Make the chart
background white

    // Compute the width of each bar
    var barwidth = Math.floor(width/data.length);
    // Find the largest number in data[]. Note the clever use of
Function.apply( ).
    var maxdata = Math.max.apply(this, data);
    // The scaling factor for the chart: scale*data[i] gives the height
of a bar
    var scale = height/maxdata;

    // Now loop through the data array and create a bar for each datum
    for(var i = 0; i < data.length; i++) {
bar
        var bar = document.createElement("div"); // Create div for
bar
        var barheight = data[i] * scale; // Compute height of
bar
        bar.style.position = "absolute"; // Set bar position
and size
        bar.style.left = (barwidth*i+1+10)+"px"; // Add bar border
and chart pad
        bar.style.top = height-barheight+10+"px"; // Add chart padding
        bar.style.width = (barwidth-2) + "px"; // -2 for bar border
        bar.style.height = (barheight-1) + "px"; // -1 for bar top
border
        bar.style.border = "solid black 1px"; // Bar border style
        bar.style.backgroundColor = barcolor; // Bar color
        bar.style.fontSize = "1px"; // IE bug workaround
        chart.appendChild(bar); // Add bar to chart
    }

    // Now add the chart we've built to the document body

```

```

    document.body.appendChild(chart);

    // Finally, return the chart element so the caller can manipulate
it
    return chart;
}

```

18.3.4 DHTML Animations

Some of the most powerful DHTML techniques you can achieve with JavaScript and CSS are animations. There is nothing particularly special about DHTML animations; all you have to do is periodically change one or more style properties of an element or elements. For example, to slide an image into place from the left, you increment the image's `style.left` property repeatedly, until it reaches the desired position. Or you can repeatedly modify the `style.clip` property to "unveil" the image pixel by pixel.

[Example 18-4](#) contains a simple HTML file that defines a `div` element to be animated and a short script that changes the background color of the element every 500 milliseconds. Note that the color change is done simply by assigning a value to a CSS style property. What makes it an animation is that the color is changed repeatedly, using the `setInterval()` function of the Window object. (You'll need to use `setInterval()` or `setTimeout()` for all DHTML animations; you may want to refresh your memory by reading about these functions in the client-side reference section.) Finally, note the use of the modulo (remainder) operator `%` to cycle through the colors. Consult [Chapter 5](#) if you've forgotten how that operator works.

Example 18-4. A simple color-changing animation

```

<!-- This div is the element we are animating -->
<div id="urgent"><h1>Red Alert!</h1>The Web server is under
attack!</div>
<!-- This is the animation script for the element -->
<script>
var e = document.getElementById("urgent");           // Get Element
object
var colors = ["white", "yellow", "orange", "red"] // Colors to cycle
through
var nextColor = 0;                                   // Position in the
cycle
// Evaluate the following expression every 500 milliseconds
// to animate the background color of the div element
setInterval("e.style.backgroundColor=colors[nextColor++%colors.length];
", 500);
</script>

```

[Example 18-4](#) produces a very simple animation. In practice, CSS animations typically involve modifications to two or more style properties (such as `top`, `left`, and `clip`) at the same time. Setting up complex animations using a technique like that shown in

[Example 18-4](#) can get quite complicated. Furthermore, in order to avoid becoming annoying, animations should typically run for a short while and then stop, but there is no way to stop the animation produced by [Example 18-4](#).

[Example 18-5](#) shows a JavaScript file that defines a CSS animation function that makes it much easier to set up animations, even complex ones. The `animateCSS()` function defined in this example is passed five arguments. The first specifies the `HTMLElement` object to be animated. The second and third arguments specify the number of frames in the animation and the length of time each frame should be displayed. The fourth argument is a JavaScript object that specifies the animation to be performed. And the fifth argument is an optional function that should be invoked once when the animation is complete.

The fourth argument to `animateCSS()` is the crucial one. Each property of the JavaScript object must have the same name as a CSS style property, and the value of each property must be a function that returns a legal value for the named style. Every time a new frame of the animation is displayed, each of these functions is called to generate a new value for each of the style properties. Each function is passed the frame number and the total elapsed time and can use these arguments to help it return an appropriate value.

An example should make the use of `animateCSS()` much clearer. The following code moves an element up the screen while gradually uncovering it by enlarging its clipping region:

```
// Animate the element with id "title" for 40 frames of 50 milliseconds each
animateCSS(document.getElementById("title"), 40, 50,
           { // Set top and clip style properties for each frame as
follows:
           top: function(f,t) { return 300-f*5 + "px"; }
           clip: function(f,t) {return "rect(auto "+f*10+"px auto
auto)";},
           });
```

The next code fragment uses `animateCSS()` to move a `Button` object in a circle. It uses the optional fifth argument to `animateCSS()` to change the button text to "Done" when the animation is complete. Note that the element being animated is passed as the argument to the function specified by the fifth argument:

```
// Move a button in a circle, then change the text it displays
animateCSS(document.forms[0].elements[0], 40, 50, // Button, 40
frames, 50ms
           { // This trigonometry defines a circle of radius 100 at
(200,200):
           left: function(f,t){ return 200 + 100*Math.cos(f/8) +
"px"},
           top: function(f,t){ return 200 + 100*Math.sin(f/8) +
"px"}
           },
```



```
function(button) { button.value = "Done"; });
```

The code in [Example 18-5](#) is fairly straightforward; all the real complexity is embedded in the properties of the animation object that you pass to `animateCSS()`, as we'll see shortly. `animateCSS()` defines a nested function called `displayNextFrame()` and does little more than use `setInterval()` to arrange for `displayNextFrame()` to be called repeatedly. `displayNextFrame()` loops through the properties of the animation object and invokes the various functions to compute the new values of the style properties.

Note that because `displayNextFrame()` is defined inside `animateCSS()`, it has access to the arguments and local variables of `animateCSS()`, even though `displayNextFrame()` is invoked after `animateCSS()` has already returned! This works even if `animateCSS()` is called more than once to animate more than one element at a time. (If you don't understand why this works, you may want to review [Section 11.4](#).)

Example 18-5. A framework for CSS-based animations

```
/**
 * AnimateCSS.js:
 * This file defines a function named animateCSS( ), which serves as a
 * framework
 * for creating CSS-based animations. The arguments to this function
 * are:
 *
 *   element: The HTML element to be animated.
 *   numFrames: The total number of frames in the animation.
 *   timePerFrame: The number of milliseconds to display each frame.
 *   animation: An object that defines the animation; described
 * below.
 *   whendone: An optional function to call when the animation
 * finishes.
 *           If specified, this function is passed element as its
 * argument.
 *
 * The animateCSS( ) function simply defines an animation framework.
 * It is
 * the properties of the animation object that specify the animation to
 * be
 * done. Each property should have the same name as a CSS style
 * property. The
 * value of each property must be a function that returns values for
 * that
 * style property. Each function is passed the frame number and the
 * total
 * amount of elapsed time, and it can use these to compute the style
 * value it
 * should return for that frame. For example, to animate an image so
 * that it
 * slides in from the upper left, you might invoke animateCSS as
 * follows:
 *
 */
```

```

* animateCSS(image, 25, 50, // Animate image for 25 frames of 50ms
each
*           { // Set top and left attributes for each frame as
follows:
*           top: function(frame,time) { return frame*8 + "px"; },
*           left: function(frame,time) { return frame*8 + "px"; }
*           });
**/
function animateCSS(element, numFrames, timePerFrame, animation,
whendone) {
    var frame = 0; // Store current frame number
    var time = 0; // Store total elapsed time

    // Arrange to call displayNextFrame( ) every timePerFrame
milliseconds.
    // This will display each of the frames of the animation.
    var intervalId = setInterval(displayNextFrame, timePerFrame);

    // The call to animateCSS( ) returns now, but the line above
ensures that
    // the nested function defined below will be invoked once for each
frame
    // of the animation. Because this function is defined inside
    // animateCSS( ), it has access to the arguments and local
variables of
    // animateCSS( ) even though it is invoked after that function has
returned!
    function displayNextFrame( ) {
        if (frame >= numFrames) { // First, see if we're
done
            clearInterval(intervalId); // If so, stop calling
ourselves
            if (whendone) whendone(element); // Invoke whendone
function
            return; // And we're finished
        }

        // Now loop through all properties defined in the animation
object
        for(var cssprop in animation) {
            // For each property, call its animation function, passing
the
            // frame number and the elapsed time. Use the return value
of the
            // function as the new value of the corresponding style
property
            // of the specified element. Use try/catch to ignore any
            // exceptions caused by bad return values.
            try {
                element.style[cssprop] = animation[cssprop](frame,
time);
            } catch(e) {}

            frame++; // Increment the frame number
            time += timePerFrame; // Increment the elapsed time

```

```
}  
}
```

18.4 DHTML in Fourth-Generation Browsers

Internet Explorer 4 and Netscape 4 were the browsers that introduced DHTML technology to the Internet. Both browsers include partial support for the CSS1 standard and support the CSS positioning attributes (which were integrated into the CSS2 standard) that are critical to DHTML. Unfortunately, the DOM standard did not exist when these fourth-generation browsers were being developed, so they do not conform to that standard. Nevertheless, it is possible to achieve DHTML effects in both browsers.

18.4.1 DHTML in Internet Explorer 4

As we saw in [Chapter 17](#), IE 4 does not support the `document.getElementById()` method, nor does it support an API for dynamically creating new nodes and inserting them into a document. Instead, it provides the `document.all[]` array as a way of locating arbitrary elements of the document and allows document content to be altered with the `innerHTML` property of document elements. IE 4 does not conform to the standards here, but it provides adequate alternatives.

Although traversing and modifying documents is an important part of DHTML, the focus of this chapter is on the dynamic use of CSS styles. The good news is that the DOM API described earlier for setting CSS style attributes through the `style` property was adopted from the IE 4 API. Thus, once you've used `document.all[]` to locate the document element you want to modify, you can script the styles of that element just as you would in a browser that fully supports the DOM API. (Remember, though, that IE 4 does not fully support CSS, so you should not expect all style properties to be scriptable.)

The CSS2 standard specifies that the `position` attribute can be used to specify absolute or relative positioning for any element in a document. IE 4 was implemented before CSS2 was complete, however, and it supports absolute positioning for only a certain subset of elements. Therefore, when using absolute positioning in IE 4, you should wrap the content you want to position or animate in `<div>` or `` tags, which do honor the CSS `position` attribute.

18.4.2 DHTML in Netscape 4

Creating DHTML effects with Netscape 4 is a more complicated affair. Netscape 4 does not support a full object model, so it does not allow JavaScript programs to refer to arbitrary HTML elements. It cannot, therefore, allow access to the inline styles of arbitrary elements. Instead, it defines a special Layer object.^[5] Any element that is absolutely positioned (that is, any element that has its `position` style set to `absolute`) is placed in a separate layer from the rest of the document. This layer can be independently

positioned, hidden, shown, lowered below or raised above other layers, and so on. The Layer API was proposed to the W3C for standardization but was never standardized. For this reason, it has been dropped by the Mozilla project and is not supported in Mozilla or in Netscape 6. Thus, the techniques described in this section are useful only in the 4.x series of browsers from Netscape.

^[5] Layers were introduced in [Chapter 17](#), when we discussed Netscape 4 compatibility with the core DOM API. Here we expand that introduction and discuss how layers provide an alternative to the core DOM API and an alternative to the DOM API for accessing CSS styles.

Each independently positioned layer in a document is represented by a Layer object, and, not surprisingly, the `layers[]` array of the Document object contains the complete set of Layer objects in a document. (Layer objects appear in this array in the order in which they appear in the document.) Additionally, any layer that is given a name with the `name` or `id` attribute can be accessed by name. For example, if a layer specifies `id="L2"`, you can refer to it in Netscape as `document.L2` or as `document.layers["L2"]`. Although Netscape 4 does not provide a way to refer to arbitrary document elements, this `layers[]` array provides a way to refer to the most important dynamic elements.

A layer is something like a separate window or frame. Although the Layer object is not the same as the Window object, it does have a `document` property, just as windows and frames do. The `document` property of a Layer object refers to a Document object: each layer has its own totally independent HTML document. Layers can even be nested; we can output some HTML text into a nested layer with code like this:

```
document.layers[1].document.layers[0].document.write("Layers Are  
Fun! ");  
document.layers[1].document.layers[0].document.close( );
```

Netscape 4 does not allow us to create or manipulate the nodes of the document tree, and it does not even support the `innerHTML` property of Internet Explorer. However, the fact that layers contain independent documents does provide a technique for dynamically modifying document content.

Although Netscape 4 defines a layer as an element with the CSS `position` style set, it does not define any way to script the styles of a layer element directly. Instead, the Layer object defines properties and methods that we can use to dynamically position layers.

The properties of the Layer object have names that are similar to important CSS style attributes, but these layer properties are not exactly the same as style properties. For example, the `left` and `top` properties of the Layer object specify the pixel position of the layer; setting these properties of a layer is like setting the `left` and `top` style properties of an element, except that the Layer properties expect numeric pixel values instead of strings that include a numeric value and a unit specification. The `visibility` property of a layer specifies whether the contents of the layer should be visible; it is a lot like the style property with the same name, except that it expects a value of `show` or `hide` instead of the CSS standard `visible` or `hidden`. The Layer object also supports a `zIndex`

property that works just like the `zIndex` style property. [Table 18-3](#) lists key CSS style properties and the Layer property that is most closely equivalent to each. Note that these are the only style properties that Netscape 4 allows to be scripted.

Table 18-3. Layer properties in Netscape 4		
CSS property	Equivalent Layer property	Layer notes
<code>left, top</code>	<code>left, top</code>	Specify pixels without units. See also <code>moveTo()</code> and <code>moveBy()</code> .
<code>zIndex</code>	<code>zIndex</code>	See also <code>moveAbove()</code> , <code>moveBelow()</code> .
<code>visibility</code>	<code>visibility</code>	Layer returns <code>show</code> or <code>hide</code> even if you set this property to the standard <code>visible</code> and <code>hidden</code> values.
<code>clip</code>	<code>clip.bottom,</code> <code>clip.height, clip.left,</code> <code>clip.right,</code> <code>clip.top, clip.width</code>	Specify pixels without units.
<code>backgroundColor</code>	<code>bgColor</code>	
<code>backgroundImage</code>	<code>background.src</code>	Set to a URL string.

As you can see from [Table 18-3](#), the Layer object supports a couple of useful properties that are not related to dynamic positioning. The `background.src` property specifies a background image for the layer, and the `bgColor` property specifies a background color for the layer. These properties correspond to the `backgroundImage` and `backgroundColor` style properties.

In addition to its properties, the Layer object offers a number of convenient methods. `moveBy()` and `moveTo()` move a layer by a relative amount or to an absolute position. `moveAbove()` and `moveBelow()` set the `zIndex` of a layer relative to that of some other layer. See the client-side reference section of this book for a complete list of Layer properties and methods.

Because every layer contains an independent document, you can dynamically update the contents of a layer with the `open()`, `write()`, and `close()` methods of the Document object, as we saw in [Chapter 14](#). In addition, the `src` property of a layer specifies the URL of the document that it displays. By setting this property, you can force the browser to load an entirely new document for display in the layer. The `load()` method is similar; it loads a new URL and changes the layer's width at the same time. Because layers often

contain dynamically generated content, you may find it convenient to use `javascript:` URLs with the `src` property and `load()` method.

We've seen that Netscape 4 automatically creates a `Layer` object for any element that has its `position` style property set to `absolute`. The Netscape 4 API also allows layers to be created in other, less standards-compliant ways. For example, Netscape 4 defines an HTML `<layer>` tag that allows layers to be defined directly in HTML. `<layer>` remains a proprietary Netscape 4 extension; it was not included in the HTML 4 standard and is not supported in Mozilla or Netscape 6. More importantly, though, Netscape 4 supports a `Layer()` constructor that allows `Layer` objects to be dynamically created, as needed within a program. See the client-side reference section of this book for details.

18.4.3 Example: A Cross-Platform DHTML Animation

Despite the differences between the DOM API, the IE 4 API, and the Netscape `Layer` API, it is still possible to create DHTML effects that work in DOM-compliant browsers, in pre-DOM versions of IE, and in Netscape 4. [Example 18-6](#) shows one way it can be done. This script displays the word "Hello" and animates it in a straight line from one point in the browser window to another.

Note the compatibility technique used in this example: we test for the existence of key functions, arrays, and properties before using them. If the `Document` object has a property named `getElementById`, we assume that we have a DOM-compliant browser with that property referring to the `getElementById()` method. Similarly, if the `Document` object has a property named `all`, we assume that we're running in Internet Explorer and use the `document.all[]` array to locate the element to be animated.

Example 18-6. A cross-browser DHTML animation script

```
<!-- This is the dynamic element we will animate. We wrap the h1 tag in
a -->
<!-- div because IE 4 won't move the h1 without a div or a span
container. -->
<div id="title" style="position:absolute"><h1>Hello</h1></div>

<!-- This is the JavaScript code that performs the animation -->
<script>
// These variables set the parameters for our animation:
var id = "title"; // Name of the element to animate
var numFrames = 30; // How many frames to display
var interval = 100; // How long to display each frame
var x0 = 100, y0 = 100; // The element's starting position
var x1 = 500, y1 = 500; // The element's ending position
var dx = (x1 - x0)/(numFrames-1); // Distance to horizontally move
each frame
var dy = (y1 - y0)/(numFrames-1); // Distance to vertically move each
frame
var frameNum = 0; // Frame we are at now
var element = null; // The element to be animated
```

```

// First, we find the element to be animated. Use a DOM-compliant
technique
// if the browser supports it; otherwise, fall back on browser-specific
code.
if (document.getElementById) { // If this is a DOM-
compliant browser,
    element = document.getElementById(id); // use the DOM method
}
else if (document.all) { // Otherwise, if the IE API is
supported,
    element = document.all[id]; // use the all[] array to find the
element
}
else if (document.layers) { // Else, if the Netscape API is
supported,
    element = document.layers[id]; // use the layers[] array to get
the element
}

// If we found the element to animate using one of the previous
techniques,
// start animating it by calling nextFrame( ) every interval
milliseconds
if (element) {
    var intervalId = setInterval("nextFrame( )", interval);
}

// This function is repeatedly called to display each frame of the
animation.
// It moves the element using either the DOM API for setting CSS style
// properties or, if the browser does not support that API, the
Netscape
// Layer API.
function nextFrame( ) {
    if (element.style) {
        // If the browser supports it, move the element by setting CSS
// style properties. Note the inclusion of the units string.
        element.style.left = x0 + dx*frameNum + "px";
        element.style.top = y0 + dy*frameNum + "px";
    }
    else {
        // Otherwise, assume that element is a layer, and move it by
// setting its properties. We could also use element.moveTo(
).
        element.left = x0 + dx*frameNum;
        element.top = y0 + dy*frameNum;
    }

    // Increment the frame number, and stop if we've reached the end
    if (++frameNum >= numFrames) clearInterval(intervalId);
}
</script>

```

18.5 Other DOM APIs for Styles and Style Sheets

So far in this chapter, we've discussed a simple DOM API for working with CSS styles: every `HTMLElement` in a document has a `style` property that represents the inline style attributes of that element. The `style` property refers to a `CSS2Properties` object that defines a JavaScript property for each CSS style attribute defined by the CSS2 standard.

Although we've made extensive use of it, the `CSS2Properties` object is just one part of the DOM API for CSS.^[6] This section provides a quick overview of the rest of the DOM API for working with CSS style sheets. Note, however, that at the time of this writing, much of the CSS API is not well supported by current (sixth-generation) browsers. You should test carefully before relying on any of the APIs described here.

^[6] In fact, the `CSS2Properties` object is optional. A DOM implementation may support CSS without supporting `CSS2Properties`. In practice, however, this is the most commonly used API for working with styles, and DOM implementations in web browsers are effectively required to support it.

18.5.1 Style Declarations

The `CSS2Properties` interface is a subinterface of `CSSStyleDeclaration`. Thus, the `style` property of each document element also implements the properties and methods of `CSSStyleDeclaration`. The methods include `setProperty()` and `getPropertyValue()`, which you can use as an alternative to setting and querying the individual style properties of `CSS2Properties`. For example, these two lines of code accomplish the same thing:

```
element.style.fontFamily = "sans-serif";
element.style.setProperty("font-family", "sans-serif", "");
```

Other features of the `CSSStyleDeclaration` interface are the `removeProperty()` method, which deletes a named style, and the `cssText` property, which returns a text representation of all the style attributes and their values. Since `CSSStyleDeclaration` objects represent a set of style attributes and their values, they can also be used as arrays to iterate through the names of the style attributes.

18.5.2 Computed Styles

As I emphasized earlier in this chapter, the `style` property of a document element represents the `style` attribute for that element, and it does not contain any information about other styles (from style sheets) that affect that element. To determine the complete set of styles that apply to an element, use the `getComputedStyle()` method of the `Window` object (this method is defined by the `AbstractView` interface: see the "`AbstractView.getComputedStyle()`" entry in the DOM reference section). The return value of this method is a `CSSStyleDeclaration` object that describes all the styles that apply to the specified element. You can assume that the returned object also implements the `CSS2Properties` interface, just as the `style` property of an element does.

To illustrate the difference between an element's inline style and its computed style, consider an element `e`. To determine whether `e` has a font specified in its inline `style` attribute, you can do this:

```
var inlinefont = e.style.fontFamily;
```

But to determine what font family `e` is displayed in (regardless of whether this is specified by an inline style or by a style sheet), do this instead:

```
var fontfamily = window.getComputedStyle(e, null).fontFamily;
```

You may prefer to use `getComputedStyle()` in a way that makes it clearer that it is defined by the `AbstractView` interface:

```
var fontfamily = document.defaultView.getComputedStyle(e, null).fontFamily;
```

The style values returned by `getComputedStyle()` are read-only, since they come from various places in the style cascade. Setting any of the attributes has no effect on the style of the element. The `getComputedStyle()` method should also be considered "expensive," since it must traverse the entire cascade and build a large `CSSStyleDeclaration` representing the many style attributes that apply to the element.

Finally, note that IE 5 and later define a nonstandard but useful `currentStyle` property in addition to the `style` property for all HTML elements. `currentStyle` refers to a `CSS2Properties` object that holds the computed style for that element.

18.5.3 Override Styles

The CSS standard specifies that a web browser have a default style sheet that defines the basic display styles of HTML elements. The browser may allow the user to specify a user style sheet that represents the user's style preferences and overrides styles specified in the default style sheet. Author style sheets are style sheets defined by a document's author -- that is, the styles included in or linked into a document. Author style sheets override the browser's default styles and the user's styles (except for `!important` styles). Inline styles specified with the `style` attribute of an element can be considered part of the author style sheet.

The DOM standard introduces the notion of an override style sheet that overrides the author style sheet, including inline styles. By setting styles for an element in the override style sheet, you can change the displayed style of an element without modifying the document's style sheets or the inline style of that element. To obtain the override style of an element, use the `getOverrideStyle()` method of the `Document` object:

```
var element = document.getElementById("title");
var override = document.getOverrideStyle(element, null);
```

This method returns a `CSSStyleDeclaration` object (which also implements `CSS2Properties`) that you can use to change the displayed style of an element. Note the difference between setting an override style and an inline style:

```
override.backgroundColor = "yellow"; // Sets an override style
element.style.backgroundColor = "pink"; // Sets an inline style
```

18.5.4 Creating Style Sheets

The `DOMImplementation` object (accessed as `document.implementation`) defines a `createCSSStyleSheet()` method for creating `CSSStyleSheet` objects. The `CSSStyleSheet` object defines an `insertRule()` method that you can use to add style rules to the style sheet. Unfortunately, DOM Level 2 does not define any way to associate a created style sheet with a document, so there is currently no point in using this method. Future versions of the DOM standard may remedy this.

18.5.5 Traversing Style Sheets

The core DOM API makes it possible to traverse an HTML (or XML) document and examine every element, attribute, and Text node of the document. Similarly, the style sheets and CSS modules of the DOM make it possible to examine all the style sheets in or linked into a document and to traverse those style sheets, examining all the rules, selectors, and style attributes that comprise them.

For scripters who want to create DHTML, it is usually sufficient simply to work within the inline styles of elements using the API shown earlier in this chapter, and it is not typically necessary to traverse style sheets. Nevertheless, this section briefly introduces the DOM API for style-sheet traversal. You can find further details on the API in the DOM reference section. At the time of this writing this API is not well supported, but support in Mozilla is expected soon. Note also that IE 5 defines a proprietary and incompatible API for traversing style sheets.

The style sheets that are included in or linked into a document are accessible through the `document.styleSheets[]` array. For example:

```
var ss = document.styleSheets[0];
```

The elements of this array are `StyleSheet` objects. `StyleSheet` represents a generic style sheet. In HTML documents using CSS style sheets, these objects all implement the subinterface `CSSStyleSheet`, which provides CSS-specific properties and methods. A

CSSStyleSheet object has a `cssRules[]` array that contains the rules of the style sheet. For example:

```
var firstRule = document.styleSheets[0].cssRules[0]
```

The CSSStyleSheet interface also defines `insertRule()` and `deleteRule()` methods for adding and removing rules from the style sheet:

```
document.styleSheets[0].insertRule("h1 { text-weight: bold; }", 0);
```

The elements of the `CSSStyleSheet.cssRules[]` array are `CSSRule` objects. CSS style sheets may contain a number of different types of rules. In addition to the basic style rules that we've seen in this chapter, there are various "at-rules," which are specified with keywords like `@import` and `@page`. You can read about these special types of CSS rules in a CSS reference.

The `CSSRule` interface is a generic one that can represent any type of rule and has subinterfaces that represent the specific rule types. The `type` property of `CSSRule` specifies the specific rule type. Most rules in a CSS style sheet are basic style rules, such as:

```
h1 { font-family: sans-serif; font-weight: bold; font-size: 24pt; }
```

Rules of this type have a `type` property of `CSSRule.STYLE_RULE` and are represented by `CSSRule` objects that additionally implement the `CSSStyleRule` interface. `CSSStyleRule` objects define a `selectorText` property that contains the rule selector (the string "h1" in the previous rule) and a `style` property that contains the rule's style attributes and values (such as the font attributes in the previous rule). For example:

```
var rule = document.styleSheets[0].cssRules[0]
var styles;
if (rule.type == CSSRule.STYLE_RULE) styles = rule.style;
```

The value of the `CSSStyleRule.style` property is a `CSSStyleDeclaration` object. We've already encountered this object: it is the same type of object that is used to represent the inline styles of document elements. It defines methods such as `setProperty()`, `removeProperty()`, and `getPropertyValue()`. As discussed previously, `CSSStyleDeclaration` objects typically also implement the `CSS2Properties` interface and therefore define a property that corresponds to each CSS attribute.

The properties of `CSS2Properties` and the `getPropertyValue()` method of `CSSStyleDeclaration` return the values of CSS style attributes as strings. As discussed earlier in this chapter, this means that when you query the value of an attribute such as

`font-size` (or when you read the `fontSize` property of `CSS2Properties`), what you get back is a number and a units value. This might be "24pt" or a (probably less useful) value like "10mm". In general, when you get the value of a CSS attribute as a string, you have to parse it in some way to extract the data you want from it. This is particularly true of attributes like `clip`, which have a complex string syntax.

As an alternative to parsing strings, `CSSStyleDeclaration` provides another method, `getPropertyCSSValue()`, that returns the value of a CSS attribute as a `CSSValue` object instead of a string. The `cssValueType` property of the `CSSValue` object specifies a sub-interface that the object also implements. If an attribute has more than one value, the `CSSValue` object implements `CSSValueList` and behaves like an array of `CSSValue` objects. Otherwise, the `CSSValue` object is typically a "primitive" value and implements the `CSSPrimitiveValue` interface. `CSSPrimitiveValue` objects have a property named `primitiveType` that specifies the type of the value or the units that apply to the value. There are 26 possible types, all represented by constants like `CSSPrimitiveValue.CSS_PERCENTAGE`, `CSSPrimitiveValue.CSS_PX`, and `CSSPrimitiveValue.CSS_RGBCOLOR`. In addition to the `primitiveType` property and the various type constants, `CSSPrimitiveValue` defines various methods for setting and querying the value represented by the object. If the `CSSPrimitiveValue` object represents a length or percentage, for example, you call `getFloatValue()` to obtain the length. If the `primitiveType` property indicates that the value represents a color, you use `getRGBColorValue()` to query the color value.

Finally, the DOM CSS API also defines a few special object types to represent attribute values: `RGBColor` objects represent color values, `Rect` objects represent rectangle values (such as the value of the `clip` attribute), and `Counter` objects represent CSS2 counters. See the DOM reference section for details.

Chapter 19. Events and Event Handling

As we saw in [Chapter 12](#), interactive JavaScript programs use an event-driven programming model. In this style of programming, the web browser generates an *event* whenever something interesting happens to the document or to some element of it. For example, the web browser generates an event when it finishes loading a document, when the user moves the mouse over a hyperlink, or when the user clicks on the Submit button of a form. If a JavaScript application cares about a particular type of event for a particular document element, it can register an *event handler* -- a JavaScript function or snippet of code -- for that type of event on the element of interest. Then, when that particular event occurs, the browser invokes the handler code. All applications with graphical user interfaces are designed this way: they sit around waiting for the user to do something interesting (i.e., they wait for events to occur) and then they respond.

As an aside, it is worth noting that timers and error handlers (both of which are described in [Chapter 13](#)) are related to the event-driven programming model. Like the event handlers described in this chapter, timers and error handlers work by registering a function with the browser and allowing the browser to call that function when the appropriate event occurs. In these cases, however, the event of interest is the passage of a specified amount of time or the occurrence of a JavaScript error. Although timers and error handlers are not discussed in this chapter, it is useful to think of them as related to event handling, and I encourage you to reread [Section 13.4](#), and [Section 13.5](#), in the context of this chapter.

Most nontrivial JavaScript programs rely heavily on event handlers. We've already seen a number of JavaScript examples that use simple event handlers. This chapter fills in all the missing details about events and event handling. Unfortunately, these details are more complex than they ought to be, because there are four distinct and incompatible event-handling models in use. These models are:

The original event model. This is the simple event-handling scheme that we've used (but not thoroughly documented) so far in this book. It was codified, to a limited extent, by the HTML 4 standard, and is informally considered to be part of the DOM Level 0 API. Although its features are limited, it is supported by all JavaScript-enabled web browsers and is therefore portable.

The standard event model. This powerful and full-featured event model was standardized by the DOM Level 2 standard. It is supported by the Netscape 6 and Mozilla browsers.

The Internet Explorer event model. This event model is implemented by IE 4 and later and has some, but not all, of the advanced features of the standard event model. Although Microsoft participated in the creation of the DOM Level 2 event model and had plenty of

time to implement this standard event model in IE 5.5 and IE 6, they have stuck with their proprietary event model instead. This means that JavaScript programmers who want to use advanced event-handling features must write special code for IE browsers.

The Netscape 4 event model. This event model was implemented in Netscape 4 and continues to be (mostly, but not fully) supported in Netscape 6, although it has been superseded by the standard event model. It has some, but not all, of the advanced features of the standard event model. JavaScript programmers who want to use advanced event-handling features and retain compatibility with Netscape 4 need to understand this model.

The rest of this chapter documents each of these event models in turn.

19.1 Basic Event Handling

In the code we've seen so far in this book, event handlers have been written as strings of JavaScript code that are used as the values of certain HTML attributes, such as `onclick`. Although this is the key to the original event model, there are a number of additional details, described in the following sections, that you should understand.

19.1.1 Events and Event Types

Different types of occurrences generate different types of events. When the user moves the mouse over a hyperlink, it causes a different type of event than when the user clicks the mouse on the **Submit** button of a form. Even the same occurrence can generate different types of events based on context: when the user clicks the mouse over a **Submit** button, for example, it generates a different event than when the user clicks the mouse over the **Reset** button of a form.

In the original event model, an event is an abstraction internal to the web browser, and JavaScript code cannot manipulate an event directly. When we speak of an *event type* in the original event model, what we really mean is the name of the event handler that is invoked in response to the event. In this model, event-handling code is specified using the attributes of HTML elements (and the corresponding properties of the associated JavaScript objects). Thus, if your application needs to know when the user moves the mouse over a specific hyperlink, you use the `onmouseover` attribute of the `<a>` tag that defines the hyperlink. And if the application needs to know when the user clicks the **Submit** button, you use the `onclick` attribute of the `<input>` tag that defines the button or the `onsubmit` attribute of the `<form>` element that contains that button.

There are quite a few different event handler attributes that you can use in the original event model. They are listed in [Table 19-1](#), which also specifies when these event handlers are triggered and which HTML elements support the handler attributes.

As client-side JavaScript programming has evolved, so has the event model it supports. With each new browser version, new event handler attributes have been added. Finally, the HTML 4 specification codified a standard set of event handler attributes for HTML

tags. The third column of [Table 19-1](#) specifies which HTML elements support each event handler attribute, and it also specifies which browser versions support that event handler for that tag and whether the event handler is a standard part of HTML 4 for that tag. In this third column, "N" is an abbreviation for Netscape and "IE" is an abbreviation for Internet Explorer. Each browser version is backward compatible with previous versions, so "N3," for example, means Netscape 3 and all later versions.

If you study the various event handler attributes in [Table 19-1](#) closely, you can discern two broad categories of events. One category is *raw events* or *input events*. These are the events that are generated when the user moves or clicks the mouse or presses a key on the keyboard. These low-level events simply describe a user's gesture and have no other meaning. The second category of events are *semantic events*. These higher-level events have a more complex meaning and can typically occur only in specific contexts: when the browser has finished loading the document or a form is about to be submitted, for example. A semantic event often occurs as a side effect of a lower-level event. For example, when the user clicks the mouse over a **Submit** button, three input handlers are triggered: `onmousedown`, `onmouseup`, and `onclick`. And, as a result of this mouse-click, the HTML form that contains the button generates an `onsubmit` event.

One final note about [Table 19-1](#) is required. For raw mouse event handlers, column three specifies that the handler attribute is supported (in HTML 4, at least) by "most elements." The HTML elements that do not support these event handlers are typically elements that belong in the `<head>` of a document or do not have a graphical representation of their own. The tags that do not support the nearly universal mouse event handler attributes are: `<applet>`, `<bdo>`, `
`, ``, `<frame>`, `<frameset>`, `<head>`, `<html>`, `<iframe>`, `<isindex>`, `<meta>`, and `<style>`.

Table 19-1. Event handlers and the HTML elements that support them

Handler	Triggered when	Supported by
<code>onabort</code>	Image loading interrupted.	N3, IE4: <code></code>
<code>onblur</code>	Element loses input focus.	HTML4, N2, IE3: <code><button></code> , <code><input></code> , <code><label></code> , <code><select></code> , <code><textarea></code> N3, IE4: <code><body></code>
<code>onchange</code>	Selection in a <code><select></code> element or other form element loses focus and its value has changed since it gained focus.	HTML4, N2, IE3: <code><input></code> , <code><select></code> , <code><textarea></code>
<code>onclick</code>	Mouse press and release; follows <code>mouseup</code> event. Return <code>false</code> to cancel default action (i.e., follow link, reset, submit).	N2, IE3: <code><a></code> , <code><area></code> , <code><input></code> HTML4, N6, IE4: most

Table 19-1. Event handlers and the HTML elements that support them

Handler	Triggered when	Supported by
		elements
<code>ondblclick</code>	Double-click.	HTML4, N6, IE4: most elements
<code>onerror</code>	Error when loading image.	N3, IE4: <code></code>
<code>onfocus</code>	Element gains input focus.	HTML4, N2, IE3: <code><button></code> , <code><input></code> , <code><label></code> , <code><select></code> , <code><textarea></code> N3, IE4: <code><body></code>
<code>onkeydown</code>	Key pressed down. Return <code>false</code> to cancel.	N4: <code><input></code> , <code><textarea></code> HTML4, N6, IE4: form elements and <code><body></code>
<code>onkeypress</code>	Key pressed and released. Return <code>false</code> to cancel.	N4: <code><input></code> , <code><textarea></code> HTML4, N6, IE4: form elements and <code><body></code>
<code>onkeyup</code>	Key released.	N4: <code><input></code> , <code><textarea></code> HTML4, N6, IE4: form elements and <code><body></code>
<code>onload</code>	Document load complete.	HTML4, N2, IE3: <code><body></code> , <code><frameset></code> N3, IE4: <code></code> N6, IE4: <code><iframe></code> , <code><object></code>
<code>onmousedown</code>	Mouse button pressed.	N4: <code><a></code> , <code><area></code> , <code></code> HTML4, N6, IE4: most elements
<code>onmousemove</code>	Mouse moved.	HTML4, N6, IE4: most elements
<code>onmouseout</code>	Mouse moves off element.	N3: <code><a></code> , <code><area></code>

Table 19-1. Event handlers and the HTML elements that support them

Handler	Triggered when	Supported by
		HTML4, N6, IE4: most elements
<code>onmouseover</code>	Mouse moves over element. For links, return <code>true</code> to prevent URL from appearing in status bar.	N2, IE3: <code><a></code> , <code><area></code> HTML4, N6, IE4: most elements
<code>onmouseup</code>	Mouse button released.	N4: <code><a></code> , <code><area></code> , <code></code> HTML4, N6, IE4: most elements
<code>onreset</code>	Form reset requested. Return <code>false</code> to prevent reset.	HTML4, N3, IE4: <code><form></code>
<code>onresize</code>	Window size changes.	N4, IE4: <code><body></code> , <code><frameset></code>
<code>onselect</code>	Text selected.	HTML4, N6, IE3: <code><input></code> , <code><textarea></code>
<code>onsubmit</code>	Form submission requested. Return <code>false</code> to prevent submission.	HTML4, N3, IE4: <code><form></code>
<code>onunload</code>	Document or frameset unloaded.	HTML4, N2, IE3: <code><body></code> , <code><frameset></code>

19.1.2 Event Handlers as Attributes

As we've seen in a number of examples prior to this chapter, event handlers are specified (in the original event model) as strings of JavaScript code used for the values of HTML attributes. So, for example, to execute JavaScript code when the user clicks a button, specify that code as the value of the `onclick` attribute of the `<input>` tag:

```
<input type="button" value="Press Me" onclick="alert('thanks');">
```

The value of an event handler attribute is an arbitrary string of JavaScript code. If the handler consists of multiple JavaScript statements, the statements *must* be separated from each other by semicolons. For example:

```
<input type="button" value="Click Here"
  onclick="if (window.numclicks) numclicks++; else numclicks=1;
  this.value='Click # ' + numclicks;">
```

When an event handler requires multiple statements, it is usually easier to define them in the body of a function and then use the HTML event handler attribute to invoke that function. For example, if you want to validate a user's form input before submitting the form, you can use the `onsubmit` attribute of the `<form>` tag. Form validation typically requires several lines of code, at a minimum, so instead of cramming all this code into one long attribute value, it makes more sense to define a form-validation function and simply use the `onclick` attribute to invoke that function. For example, if you defined a function named `validateForm()` to perform validation, you could invoke it from an event handler like this:

```
<form action="processform.cgi" onsubmit="return validateForm( );">
```

Remember that HTML is case-insensitive, so you can capitalize event handler attributes any way you choose. One common convention is to use mixed-case capitalization, with the initial "on" prefix in lowercase: `onClick`, `onLoad`, `onMouseOut`, and so on. In this book, I've chosen to use all lowercase, however, for compatibility with XHTML, which is case-sensitive.

The JavaScript code in an event handler attribute may contain a `return` statement, and the return value may have special meaning to the browser. This is discussed shortly. Also, note that the JavaScript code of an event handler runs in a different scope (see [Chapter 4](#)) than global JavaScript code. This, too, is discussed in more detail later in this section.

19.1.3 Event Handlers as Properties

We've seen that each HTML element in a document has a corresponding JavaScript object in the document tree, and the properties of this JavaScript object correspond to the attributes of the HTML element. In JavaScript 1.1 and later, this applies to event handler attributes as well. So if an `<input>` tag has an `onclick` attribute, the event handler it contains can be referred to with the `onclick` property of the form element object. (JavaScript is case-sensitive, so regardless of the capitalization used for the HTML attribute, the JavaScript property must be all lowercase.)

Technically, the DOM specification does not support the original event model we've described here and does not define JavaScript attributes that correspond to the event handler attributes standardized by HTML 4. Despite the lack of formal standardization by the DOM, this event model is so widely used that all JavaScript-enabled web browsers allow event handlers to be referred to as JavaScript properties.

Since the value of an HTML event handler attribute is a string of JavaScript code, you might expect the value of the corresponding JavaScript property to be a string as well.

This is not the case: when accessed through JavaScript, event handler properties are functions. You can verify this with a simple example:

```
<input type="button" value="Click Here" onclick="alert(typeof
this.onclick);">
```

If you click the button, it displays a dialog box containing the word "function," not the word "string." (Note that in event handlers, the `this` keyword refers to the object on which the event occurred. We'll discuss the `this` keyword shortly.)

To assign an event handler to a document element using JavaScript, simply set the event handler property to the desired function. For example, consider the following HTML form:

```
<form name="f1">
<input name="b1" type="button" value="Press Me">
</form>
```

The button in this form can be referred to as `document.f1.b1`, which means that an event handler can be assigned with a line of JavaScript like this one:

```
document.f1.b1.onclick=function( ) { alert('Thanks!'); };
```

An event handler can also be assigned like this:

```
function plead( ) { window.status = "Please Press Me!"; }
document.f1.b1.onmouseover = plead;
```

Pay particular attention to that last line: there are no parentheses after the name of the function. To define an event handler, we are assigning the function itself to the event handler property, not the result of invoking the function. This is an area that often trips up beginning JavaScript programmers.

There are a couple of advantages to expressing event handlers as JavaScript properties. First, it reduces the intermingling of HTML and JavaScript, promoting modularity and cleaner, more maintainable code. Second, it allows event handler functions to be dynamic. Unlike HTML attributes, which are a static part of the document and can be set only when the document is created, JavaScript properties can be changed at any time. In complex interactive programs, it can sometimes be useful to dynamically change the event handlers registered for HTML elements. One minor disadvantage to defining event handlers in JavaScript is that it separates the handler from the element to which it belongs. If the user interacts with a document element before the document is fully loaded (and before all its scripts have executed), the event handlers for the document element may not yet be defined.

[Example 19-1](#) shows how you can specify a single function to be the event handler for many document elements. The example is a simple function that defines an `onclick` event handler for every link in a document. The event handler asks for the user's confirmation before allowing the browser to follow the hyperlink on which the user has just clicked. The event handler function returns `false` if the user does not confirm, which prevents the browser from following the link. Event handler return values will be discussed shortly.

Example 19-1. One function, many event handlers

```
// This function is suitable for use as an onclick event handler for
<a> and
// <area> elements. It uses the this keyword to refer to the document
element
// and may return false to prevent the browser from following the link.
function confirmLink( ) {
    return confirm("Do you really want to visit " + this.href + "?");
}

// This function loops through all the hyperlinks in a document and
assigns
// the confirmLink function to each one as an event handler. Don't call
it
// before the document is parsed and the links are all defined. It is
best
// to call it from the onload event handler of a <body> tag.
function confirmAllLinks( ) {
    for(var i = 0; i < document.links.length; i++) {
        document.links[i].onclick = confirmLink;
    }
}
```

19.1.3.1 Explicitly invoking event handlers

Because the values of JavaScript event handler properties are functions, we can use JavaScript to invoke event handler functions directly. For example, if we've used the `onsubmit` attribute of a `<form>` tag to define a form-validation function and we want to validate the form at some point before the user attempts to submit it, we can use the `onsubmit` property of the Form object to invoke the event handler function. The code might look like this:

```
document.myform.onsubmit( );
```

Note, however, that invoking an event handler is not a way to simulate what happens when the event actually occurs. If we invoke the `onclick` method of a Link object, for example, it does not make the browser follow the link and load a new document. It merely executes whatever function we've defined as the value of that property. To make the browser load a new document, we have to set the `location` property of the Window object, as we saw in [Chapter 13](#). The same is true of the `onsubmit` method of a Form

object or the `onclick` method of a Submit object: invoking the method runs the event handler function but does not cause the form to be submitted. (To actually submit the form, we call the `submit()` method of the Form object.)

One reason that you might want to explicitly invoke an event handler function is if you want to use JavaScript to augment an event handler that is (or may be) already defined by HTML code. Suppose you want to take a special action when the user clicks a button, but you do not want to disrupt any `onclick` event handler that may have been defined in the HTML document itself. (This is one of the problems with the code in [Example 19-1](#) -- by adding a handler for each hyperlink, it overwrites any `onclick` handlers that were already defined for those hyperlinks.) You might accomplish this with code like the following:

```
var b = document.myform.mybutton; // This is the button we're
interested in
var oldHandler = b.onclick; // Save the HTML event handler
function newHandler( ) { /* My event-handling code goes here */ }
// Now assign a new event handler that calls both the old and new
handlers
b.onclick = function( ) { oldHandler(); newHandler( ); }
```

19.1.4 Event Handler Return Values

In many cases, an event handler (whether specified by HTML attribute or JavaScript property) uses its return value to indicate the disposition of the event. For example, if you use the `onsubmit` event handler of a Form object to perform form validation and discover that the user has not filled in all the fields, you can return `false` from the handler to prevent the form from actually being submitted. You can ensure that a form is not submitted with an empty text field like this:

```
<form action="search.cgi"
      onsubmit="if (this.elements[0].value.length == 0) return false;">
<input type="text">
</form>
```

Generally, if the web browser performs some kind of default action in response to an event, you can return `false` to prevent the browser from performing that action. In addition to `onsubmit`, other event handlers from which you can return `false` to prevent the default action include `onclick`, `onkeydown`, `onkeypress`, `onmousedown`, `onmouseup`, and `onreset`. The second column of [Table 19-1](#) contains a note about the return values for these event handlers.

There is one exception to the rule about returning `false` to cancel: when the user moves the mouse over a hyperlink (or image map), the browser's default action is to display the link's URL in the status line. To prevent this from happening, you must return `true` from the `onmouseover` event handler. For example, you can display a message other than a URL with code like this:

```
<a href="help.htm" onmouseover="window.status='Help!!!'; return true;">Help</a>
```

There is no good reason for this exception: it is this way simply because that is always the way it has been.

Note that event handlers are never required to explicitly return a value. If you don't return a value, the default behavior occurs.

19.1.5 Event Handlers and the `this` Keyword

Whether you define an event handler with an HTML attribute or with a JavaScript property, what you are doing is assigning a function to a property of a document element. In other words, you're defining a new method of the document element. When your event handler is invoked, it is invoked as a method of the element on which the event occurred, so the `this` keyword refers to that target element. This behavior is useful and unsurprising.

Be sure, however, that you understand the implications. Suppose you have an object `o` with a method `mymethod`. You might register an event handler like this:

```
button.onclick= o.mymethod;
```

This statement makes `button.onclick` refer to the same function that `o.mymethod` does. This function is now a method of both `o` and `button`. When the browser triggers this event handler, it invokes the function as a method of the `button` object, not as a method of `o`. The `this` keyword refers to the `Button` object, not to your object `o`. Do not make the mistake of thinking you can trick the browser into invoking an event handler as a method of some other object. If you want to do that, you must do it explicitly, like this:

```
button.onclick = function() { o.mymethod( ); }
```

19.1.6 Scope of Event Handlers

As we discussed in [Chapter 11](#), functions in JavaScript are lexically scoped. This means that they run in the scope in which they were defined, not in the scope from which they are called. When you define an event handler by setting the value of an HTML attribute to a string of JavaScript code, you are implicitly defining a JavaScript function (as you can see when you examine the type of the corresponding event handler property in JavaScript). It is important to understand that the scope of an event handler function defined in this way is not the same as the scope of other normally defined global JavaScript functions. This means that event handlers defined as HTML attributes execute in a different scope than other functions.¹¹

^[1] It is important to understand this, and while the discussion that follows is interesting, it is also dense. You may want to skip it on your first time through this chapter and come back to it later.

Recall from the discussion in [Chapter 4](#) that the scope of a function is defined by a scope chain, or list of objects, that is searched, in turn, for variable definitions. When a variable `x` is looked up or resolved in a normal function, JavaScript first looks for a local variable or argument by checking the call object of the function for a property of that name. If no such property is found, JavaScript proceeds to the next object in the scope chain: the global object. It checks the properties of the global object to see if the variable is a global variable.

Event handlers defined as HTML attributes have a more complex scope chain than this. The head of the scope chain is the call object. Any arguments passed to the event handler are defined here (we'll see later in this chapter that in some advanced event models, event handlers are passed an argument), as are any local variables declared in the body of the event handler. The next object in an event handler's scope chain isn't the global object, however; it is the object that triggered the event handler. So, for example, suppose you use an `<input>` tag to define a Button object in an HTML form and then use the `onclick` attribute to define an event handler. If the code for the event handler uses a variable named `form`, that variable is resolved to the `form` property of the Button object. This can be a useful shortcut when writing event handlers as HTML attributes.

The scope chain of an event handler does not stop with the object that defines the handler: it proceeds up the containment hierarchy. For the `onclick` event handler described earlier, the scope chain begins with the call object of the handler function. Then it proceeds to the Button object, as we've discussed. After that, it continues up the HTML element containment hierarchy and includes, at a minimum, the HTML `<form>` element that contains the button and the Document object that contains the form. The precise composition of the scope chain has never been standardized and is implementation-dependent. Netscape 6 and Mozilla include all containing objects (even things such as `<div>` tags), while IE 6 sticks to a more minimal set that includes the target element, plus the containing Form object (if any) and the Document object. Regardless of the browser, the final object in the scope chain is the Window object, as it always is in client-side JavaScript.

Having the target object in the scope chain of an event handler can be a useful shortcut. But having an extended scope chain that includes other document elements can be a nuisance. Consider, for example, that both the Window and Document objects define methods named `open()`. If you use the identifier `open` without qualification, you are almost always referring to the `window.open()` method. In an event handler defined as an HTML attribute, however, the Document object is in the scope chain before the Window object, and using `open` by itself refers to the `document.open()` method. Similarly, consider what would happen if you added a property named `window` to a Form object (or defined an input field with `name="window"`). Then, if you define an event handler within the form that uses the expression `window.open()`, the identifier `window` resolves to the property of the Form object rather than the global Window object, and

event handlers within the form have no easy way to refer to the global Window object or to call the `window.open()` method!

The moral is that you must be careful when defining event handlers as HTML attributes. Your safest bet is to keep any such handlers very simple. Ideally, they should just call a global function defined elsewhere and perhaps return the result:

```
<script>
function validateForm( ) {
  /* Form validation code here */
  return true;
}
</script>
<input type="submit" onclick="return validateForm( );">
```

A simple event handler like this is still executed using an unusual scope chain, and you can subvert it by defining a `validateForm()` method on one of the containing elements. But, assuming that the intended global function does get called, that function executes in the normal global scope. Once again, remember that functions are executed using the scope in which they were defined, not the scope from which they are invoked. So, even though our `validateForm()` method is invoked from an unusual scope, it is still executed in its own global scope with no possibility for confusion.

Furthermore, since there is no standard for the precise composition of the scope chain of an event handler, it is safest to assume that it contains only the target element and the global Window object. For example, use `this` to refer to the target element, and when the target is an `<input>` element, feel free to use `form` to refer to the containing Form object. But don't rely on the Form or Document objects being in the scope chain. For example, don't use the unqualified identifier `write` to refer to the Document's `write()` method. Instead, spell out that you mean `document.write()`.

Keep in mind that this entire discussion of event-handler scope applies only to event handlers defined as HTML attributes. If you specify an event handler by assigning a function to an appropriate JavaScript event handler property, there is no special scope chain involved, and your function executes in the scope in which it was defined. This is almost always the global scope, unless it is a nested function, in which case the scope chain can get interesting again!

19.2 Advanced Event Handling with DOM Level 2

The event-handling techniques we've seen so far in this chapter are part of the Level 0 DOM: the de facto standard API that is supported by every JavaScript-enabled browser. The DOM Level 2 standard defines an advanced event-handling API that is significantly different (and quite a bit more powerful) than the Level 0 API. The Level 2 standard does not incorporate the existing API into the standard DOM, but there is no danger of the

Level 0 API being dropped. For basic event-handling tasks, you should feel free to continue to use the simple API.

The Level 2 DOM Events module is supported by Mozilla and Netscape 6, but is not supported by Internet Explorer 6.

19.2.1 Event Propagation

In the Level 0 event model, the browser dispatches events to the document elements on which they occur. If that object has an appropriate event handler, that handler is run. There is nothing more to it. The situation is more complex in the Level 2 DOM. In this advanced event model, when an event occurs on a Document node (known as the event *target*), the target's event handler or handlers are triggered, but in addition, each of the target's ancestor nodes has one or two opportunities to handle that event. Event propagation proceeds in three phases. First, during the *capturing* phase, events propagate from the Document object down through the document tree to the target node. If any of the ancestors of the target (but not the target itself) has a specially registered capturing event handler, those handlers are run during this phase of event propagation. (We'll learn how both regular and capturing event handlers are registered shortly.)

The next phase of event propagation occurs at the target node itself: any appropriate event handlers registered directly on the target are run. This is akin to the kind of event handling provided by the Level event model.

The third phase of event propagation is the *bubbling* phase, in which the event propagates or bubbles back up the document hierarchy from the target element up to the Document object. Although all events are subject to the capturing phase of event propagation, not all types of events bubble: for example, it does not make sense for a submit event to propagate up the document beyond the `<form>` element to which it is directed. On the other hand, generic events such as mousedown events can be of interest to any element in the document, so they do bubble up through the document hierarchy, triggering any appropriate event handlers on each of the ancestors of the target element. In general, raw input events bubble, while higher-level semantic events do not. (See [Table 19-3](#), later in this chapter, for a definitive list of which events bubble and which do not.)

During event propagation, it is possible for any event handler to stop further propagation of the event by calling the `stopPropagation()` method of the Event object that represents the event. We'll see more about the Event object and its `stopPropagation()` method later in this chapter.

Some events cause an associated default action to be performed by the web browser. For example, when a click event occurs on an `<a>` tag, the browser's default action is to follow the hyperlink. Default actions like these are performed only after all three phases of event propagation complete, and any of the handlers invoked during event propagation have the opportunity to prevent the default action from occurring by calling the `preventDefault()` method of the Event object.

Although this kind of event propagation may seem convoluted, it provides an important means of centralizing your event-handling code. The Level 1 DOM exposes all document elements and allows events (such as mouseover events) to occur on any of those elements. This means that there are many, many more places for event handlers to be registered than there were with the old Level 0 event model. Suppose you want to trigger an event handler whenever the user moves the mouse over a `<p>` element in your document. Instead of registering an `onmouseover` event handler for each `<p>` tag, you can instead register a single event handler on the Document object and handle these events during either the capturing or bubbling phase of event propagation.

There is one other important detail about event propagation. In the Level 0 model, you can register only a single event handler for a particular type of event for a particular object. In the Level 2 model, however, you can register any number of handler functions for a particular event type on a particular object. This applies also to ancestors of an event target whose handler function or functions are invoked during the capturing or bubbling phases of event propagation.

19.2.2 Event Handler Registration

In the Level 0 API, you register an event handler by setting an attribute in your HTML or an object property in your JavaScript code. In the Level 2 event model, you register an event handler for a particular element by calling the `addEventListener()` method of that object. (The DOM standard uses the term "listener" in its API, but we'll continue to use the synonymous word "handler" in our discussion.) This method takes three arguments. The first is the name of the event type for which the handler is being registered. The event type should be a string that contains the lowercase name of the HTML handler attribute, with the leading "on" removed. Thus, if you use an `onmousedown` HTML attribute or `onmousedown` property in the Level 0 model, you'd use the string "mousedown" in the Level 2 event model.

The second argument to `addEventListener()` is the handler (or listener) function that should be invoked when the specified type of event occurs. When your function is invoked, it is passed an Event object as its only argument. This object contains details about the event (such as which mouse button was pressed) and defines methods such as `stopPropagation()`. We'll learn more about the Event interface and its subinterfaces later.

The final argument to `addEventListener()` is a boolean value. If `true`, the specified event handler is used to capture events during the capturing phase of event propagation. If the argument is `false`, the event handler is a normal event handler and is triggered when the event occurs directly on the object or on a descendant of the element and subsequently bubbles up to the element.

For example, you might use `addEventListener()` as follows to register a handler for submit events on a `<form>` element:

```
document.myform.addEventListener("submit",
                                function(e) { validate(e.target); }
                                false);
```

Or, if you wanted to capture all mousedown events that occur within a particular named `<div>` element, you might use `addEventListener()` like this:

```
var mydiv = document.getElementById("mydiv");
mydiv.addEventListener("mousedown", handleMouseDown, true);
```

Note that these examples assume that you've defined functions named `validate()` and `handleMouseDown()` elsewhere in your JavaScript code.

Event handlers registered with `addEventListener()` are executed in the scope in which they are defined. They are not invoked with the augmented scope chain that is used for event handlers defined as HTML attributes. (See [Section 19.1.6](#).)

Because event handlers are registered in the Level 2 model by invoking a method rather than by setting an attribute or property, we can register more than one event handler for a given type of event on a given object. If you call `addEventListener()` multiple times to register more than one handler function for the same event type on the same object, all of the functions you've registered are invoked when an event of that type occurs on (or bubbles up to, or is captured by) that object. It is important to understand that the DOM standard makes no guarantees about the order in which the handler functions of a single object are invoked, so you should not rely on them being called in the order in which you registered them. Also note that if you register the same handler function more than once on the same element, all registrations after the first are ignored.

Why would you want to have more than one handler function for the same event on the same object? This can be quite useful for modularizing your software. Suppose, for example, that you've written a reusable module of JavaScript code that uses mouseover events on images to perform image rollovers. Now suppose that you have another module that wants to use the same mouseover events to display additional information about the image (or the link that the image represents) in the browser's status line. With the Level API, you'd have to merge your two modules into one, so that they could share the single `onmouseover` property of the Image object. With the Level 2 API, on the other hand, each module can register the event handler it needs without knowing about or interfering with the other module.

`addEventListener()` is paired with a `removeEventListener()` method that expects the same three arguments but removes an event handler function from an object rather than adding it. It is often useful to temporarily register an event handler and then remove it soon afterward. For example, when you get a mousedown event, you might register temporary capturing event handlers for mousemove and mouseup events so you can see if

the user drags the mouse. You'd then deregister these handlers when the mouseup event arrives. In such a situation, your event-handler removal code might look as follows:

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

Both the `addEventListener()` and `removeEventListener()` methods are defined by the `EventTarget` interface. In web browsers that support the Level 2 DOM Event API, all Document nodes implement this interface. For more information about these event-handler registration and deregistration methods, look up the `EventTarget` interface in the DOM reference section.

One final note about event-handler registration: in the Level 2 DOM, event handlers are not restricted to document elements; you can also register handlers for Text nodes. In practice, however, you may find it simpler to register handlers on containing elements and allow Text node events to bubble up and be handled at the container level.

19.2.3 `addEventListener()` and the `this` Keyword

In the original Level 0 event model, when a function is registered as an event handler for a document element, it becomes a method of that document element (as discussed previously in [Section 19.1.5](#)). When the event handler is invoked, it is invoked as a method of the element, and, within the function, the `this` keyword refers to the element on which the event occurred.

In Mozilla and Netscape 6, when you register an event handler function with `addEventListener()`, it is treated the same way: when the browser invokes the function, it invokes it as a method of the document element for which it was registered. Note, however, that this is implementation-dependent behavior, and the DOM specification does not require that this happen. Thus, you should not rely on the value of the `this` keyword in your event handler functions when using the Level 2 event model. Instead, use the `currentTarget` property of the Event object that is passed to your handler functions. As we'll see when we consider the Event object later in this chapter, the `currentTarget` property refers to the object on which the event handler was registered but does so in a portable way.

19.2.4 Registering Objects as Event Handlers

`addEventListener()` allows us to register event handler functions. As discussed in the previous section, whether these functions are invoked as methods of the objects for which they are registered is implementation-dependent. For object-oriented programming, you may prefer to define event handlers as methods of a custom object and then have them invoked as methods of that object. For Java programmers, the DOM standard allows exactly this: it specifies that event handlers are objects that implement the `EventListener` interface and a method named `handleEvent()`. In Java, when you register an event

handler, you pass an object to `addEventListener()`, not a function. For simplicity, the JavaScript binding of the DOM API does not require us to implement an `EventListener` interface and instead allows us to pass function references directly to `addEventListener()`.

If you are writing an object-oriented JavaScript program and prefer to use objects as event handlers, you might use a function like this to register them:

```
function registerObjectEventHandler(element, eventtype, listener,
captures) {
    element.addEventListener(eventtype,
                             function(event) {
listener.handleEvent(event); }
                             captures);
}
```

Any object can be registered as an event listener with this function, as long as it defines a method named `handleEvent()`. That method is invoked as a method of the listener object, and the `this` keyword refers to the listener object, not to the document element that generated the event. This function works because it uses a nested function literal to capture and remember the listener object in its scope chain. (If this doesn't make sense to you, you may want to review [Section 11.4](#).)

Although it is not part of the DOM specification, Mozilla 0.9.1 and Netscape 6.1 (but not Netscape 6.0 or 6.01) allow event listener objects that define a `handleEvent()` method to be passed directly to `addEventListener()` instead of a function reference. For these browsers, a special registration function like the one we just defined is not necessary.

19.2.5 Event Modules and Event Types

As I've noted before, the Level 2 DOM is modularized, so an implementation can support parts of it and omit support for other parts. The Events API is one such module. You can test whether a browser supports this module with code like this:

```
document.implementation.hasFeature("Events", "2.0")
```

The Events module contains only the API for the basic event-handling infrastructure, however. Support for specific types of events is delegated to submodules. Each submodule provides support for a category of related event types and defines an Event type that is passed to event handlers for each of those types. For example, the submodule named `MouseEvent` provides support for `mousedown`, `mouseup`, `click`, and related event types. It also defines the `MouseEvent` interface. An object that implements this interface is passed to the handler function for any event type supported by the module.

[Table 19-2](#) lists each event module, the event interface it defines, and the types of events it supports. Note that the Level 2 DOM does not standardize any type of keyboard event, so no module of key events is listed here. Support for this type of event is expected in the DOM Level 3 standard.

Table 19-2. Event modules, interfaces, and types		
Module name	Event interface	Event types
HTMLEvents	Event	abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload
MouseEvent	MouseEvent	click, mousedown, mousemove, mouseout, mouseover, mouseup
UIEvents	UIEvent	DOMActivate, DOMFocusIn, DOMFocusOut
MutationEvents	MutationEvent	DOMAttrModified, DOMCharacterDataModified, DOMNodeInserted, DOMNodeInsertedIntoDocument, DOMNodeRemoved, DOMNodeRemovedFromDocument, DOMSubtreeModified

As you can see from [Table 19-2](#), The HTMLEvents and MouseEvent modules define event types that are familiar from the Level 0 event module. The UIEvents module defines event types that are similar to the focus, blur, and click events supported by HTML form elements but are generalized so that they can be generated by any document element that can receive focus or be activated in some way. The MutationEvents module defines events that are generated when the document changes (is mutated) in some way. These are specialized event types and are not commonly used.

As I noted earlier, when an event occurs, its handler is passed an object that implements the Event interface associated with that type of event. The properties of this object provide details about the event that may be useful to the handler. [Table 19-3](#) lists the standard events again, but this time organizes them by event type, rather than by event module. For each event type, this table specifies the kind of event object that is passed to its handler, whether this type of event bubbles up the document hierarchy during event propagation (the "B" column), and whether the event has a default action that is cancelable with the `preventDefault()` method (the "C" column). For events in the HTMLEvents module, the fifth column of the table specifies which HTML elements can generate the event. For all other event types, the fifth column specifies which properties of the event object contain meaningful event details (these properties are documented in the next section). Note that the properties listed in this column do not include the properties that are defined by the basic Event interface, which contain meaningful values for all event types.

It is useful to compare [Table 19-3](#) with [Table 19-1](#), which lists the Level 0 event handlers defined by HTML 4. The event types supported by the two models are largely the same (excluding the UIEvents and MutationEvents modules). The DOM Level 2 standard adds support for the abort, error, resize, and scroll event types that were not standardized by HTML 4, and it does not support the dblclick event type that is part of the HTML 4 standard. (Instead, as we'll see shortly, the `detail` property of the object passed to a `click` event handler specifies the number of consecutive clicks that have occurred.)

Table 19-3. Event types				
Event type	Interface	B	C	Supported by/detail properties
abort	Event	yes	no	, <object>
blur	Event	no	no	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>
change	Event	yes	no	<input>, <select>, <textarea>
click	MouseEvent	yes	yes	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
error	Event	yes	no	<body>, <frameset>, , <object>
focus	Event	no	no	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>
load	Event	no	no	<body>, <frameset>, <iframe>, , <object>
mousedown	MouseEvent	yes	yes	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
mousemove	MouseEvent	yes	no	screenX, screenY, clientX, clientY,

Table 19-3. Event types

Event type	Interface	B	C	Supported by/detail properties
				altKey, ctrlKey, shiftKey, metaKey
mouseout	MouseEvent	yes	yes	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, relatedTarget
mouseover	MouseEvent	yes	yes	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, relatedTarget
mouseup	MouseEvent	yes	yes	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
reset	Event	yes	no	<form>
resize	Event	yes	no	<body>, <frameset>, <iframe>
scroll	Event	yes	no	<body>
select	Event	yes	no	<input>, <textarea>
submit	Event	yes	yes	<form>
unload	Event	no	no	<body>, <frameset>
DOMActivate	UIEvent	yes	yes	detail
DOMAttrModified	MutationEvent	yes	no	attrName, attrChange, prevValue, newValue, relatedNode
DOMCharacterDataModified	MutationEvent	yes	no	prevValue, newValue
DOMFocusIn	UIEvent	yes	no	none
DOMFocusOut	UIEvent	yes	no	none
DOMNodeInserted	MutationEvent	yes	no	relatedNode

Table 19-3. Event types

Event type	Interface	B	C	Supported by/detail properties
DOMNodeInsertedIntoDocument	MutationEvent	no	no	none
DOMNodeRemoved	MutationEvents	yes	no	<code>relatedNode</code>
DOMNodeRemovedFromDocument	MutationEvent	no	no	none
DOMSubtreeModified	MutationEvent	yes	no	none

19.2.6 Event Interfaces and Event Details

When an event occurs, the DOM Level 2 API provides additional details about the event (such as when and where it occurred) as properties of an object that is passed to the event handler. Each event module has an associated event interface that specifies details appropriate to that type of event. [Table 19-2](#) (earlier in this chapter) lists four different event modules and four different event interfaces.

These four interfaces are actually related to one another and form a hierarchy. The Event interface is the root of the hierarchy; all event objects implement this most basic event interface. UIEvent is a subinterface of Event: any event object that implements UIEvent also implements all the methods and properties of Event. The MouseEvent interface is a subinterface of UIEvent. This means, for example, that the event object passed to an event handler for a click event implements all the methods and properties defined by each of the MouseEvent, UIEvent, and Event interfaces. Finally, the MutationEvent interface is a subinterface of Event.

The following sections introduce each of the event interfaces and highlight their most important properties and methods. You will find complete details about each interface in the DOM reference section of this book.

19.2.6.1 Event

The event types defined by the HTMLEvents module use the Event interface. All other event types use subinterfaces of this interface, which means that Event is implemented by all event objects and provides detailed information that applies to all event types. The Event interface defines the following properties (note that these properties, and the properties of all Event subinterfaces, are read-only):

`type`

The type of event that occurred. The value of this property is the name of the event type and is the same string value that was used when registering the event handler (e.g., "click" or "mouseover").

`target`

The node on which the event occurred, which may not be the same as `currentTarget`.

`currentTarget`

The node at which the event is currently being processed (i.e., the node whose event handler is currently being run). If the event is being processed during the capturing or bubbling phase of propagation, the value of this property is different from the value of the `target` property. As discussed earlier, you should use this property instead of the `this` keyword in your event handler functions.

`eventPhase`

A number that specifies what phase of event propagation is currently in process. The value is one of the constants `Event.CAPTURING_PHASE`, `Event.AT_TARGET`, or `Event.BUBBLING_PHASE`.

`timeStamp`

A Date object that specifies when the event occurred.

`bubbles`

A boolean that specifies whether this event (and events of this type) bubbles up the document tree.

`cancelable`

A boolean that specifies whether the event has a default action associated with it that can be canceled with the `preventDefault()` method.

In addition to these seven properties, the Event interface defines two methods that are also implemented by all event objects: `stopPropagation()` and `preventDefault()`. Any event handler can call `stopPropagation()` to prevent the event from being

propagated beyond the node at which it is currently being handled. Any event handler can call `preventDefault()` to prevent the browser from performing a default action associated with the event. Calling `preventDefault()` in the DOM Level 2 API is like returning `false` in the Level 0 event model.

19.2.6.2 UIEvent

The UIEvent interface is a subinterface of Event. It defines the type of event object passed to events of type DOMFocusIn, DOMFocusOut, and DOMActivate. These event types are not commonly used; what is more important about the UIEvent interface is that it is the parent interface of MouseEvent. UIEvent defines two properties in addition to those defined by Event:

`view`

The Window object (known as a "view" in DOM terminology) within which the event occurred.

`detail`

A number that may provide additional information about the event. For click, mousedown, and mouseup events, this field is the click count: 1 for a single-click, 2 for a double-click, and 3 for a triple-click. (Note that each click generates an event, but if multiple clicks are close enough together, the `detail` value indicates that. That is, a mouse event with a `detail` of 2 is always preceded by a mouse event with a `detail` of 1.) For DOMActivate events, this field is 1 for a normal activation or 2 for a hyperactivation, such as a double-click or **Shift-Enter** combination.

19.2.6.3 MouseEvent

The MouseEvent interface inherits the properties and methods of Event and UIEvent and defines the following additional properties:

`button`

A number that specifies which mouse button changed state during a mousedown, mouseup, or click event. A value of 0 indicates the left button, 1 indicates the middle button, and 2 indicates the right button. This property is used only when a button changes state: it is not used to report whether a button is held down during a mousemove event, for example. Note also that Netscape 6 gets this wrong and uses the values 1, 2, and 3, instead of 0, 1, and 2. This problem is fixed in Netscape 6.1.

`altKey`, `ctrlKey`, `metaKey`, `shiftKey`

These four boolean fields indicate whether the **Alt**, **Ctrl**, **Meta**, or **Shift** keys were held down when a mouse event occurred. Unlike the `button` property, these key properties are valid for any type of mouse event.

`clientX`, `clientY`

These two properties specify the X and Y coordinates of the mouse pointer, relative to the client area or browser window. Note that these coordinates do not take document scrolling into account: if an event occurs at the very top of the window, `clientY` is 0, regardless of how far down the document has been scrolled. Unfortunately, the Level 2 DOM does not provide a standard way to translate these window coordinates to document coordinates. In Netscape 6, you can add `window.pageXOffset` and `window.pageYOffset`, and in Internet Explorer, you can add `document.body.scrollLeft` and `document.body.scrollTop`.

`screenX`, `screenY`

These two properties specify the X- and Y-coordinates of the mouse pointer relative to the upper-left corner of the user's monitor. These values are useful if you plan to open a new browser window at or near the location of the mouse event.

`relatedTarget`

This property refers to a node that is related to the target node of the event. For mouseover events, it is the node that the mouse left when it moved over the target. For mouseout events, it is the node that the mouse entered when leaving the target. It is unused for other event types.

19.2.6.4 MutationEvent

The `MutationEvent` interface is a subinterface of `Event`, and is used to provide event details for the event types defined by the `MutationEvents` module. These event types are not commonly used in DHTML programming, so details on the interface are not provided here. See the DOM reference section for details.

19.2.7 Example: Dragging Document Elements

Now that we've discussed event propagation, event-handler registration, and the various event object interfaces for the DOM Level 2 event model, we can finally look at how they work. [Example 19-2](#) shows a JavaScript function, `beginDrag()`, that, when invoked from a `mousedown` event handler, allows a document element to be dragged by the user.

`beginDrag()` takes two arguments. The first is the element that is to be dragged. This may be the element on which the `mousedown` event occurred or a containing element (e.g., you might allow the user to drag on the titlebar of a window to move the entire window). In either case, however, it must refer to a document element that is absolutely positioned using the CSS `position` attribute, and the `left` and `top` CSS attributes must be explicitly set to pixel values in a `style` attribute. The second argument is the event object associated with the triggering `mousedown` event.

`beginDrag()` records the position of the `mousedown` event and then registers event handlers for the `mousemove` and `mouseup` events that will follow the `mousedown` event. The handler for the `mousemove` event is responsible for moving the document element, and the handler for the `mouseup` event is responsible for deregistering itself and the `mousemove` handler. It is important to note that the `mousemove` and `mouseup` handlers are registered as capturing event handlers, because the user can move the mouse faster than the document element can follow it, and some of these events occur outside of the original target element. Also, note that the `moveHandler()` and `upHandler()` functions that are registered to handle these events are defined as functions nested within `beginDrag()`. Because they are defined in this nested scope, they can use the arguments and local variables of `beginDrag()`, which considerably simplifies their implementation.

Example 19-2. Dragging with the DOM Level 2 event model

```
/**
 * Drag.js:
 * This function is designed to be called from a mousedown event
handler.
 * It registers temporary capturing event handlers for the mousemove
and
 * mouseup events that will follow and uses these handlers to "drag"
the
 * specified document element. The first argument must be an absolutely
 * positioned document element. It may be the element that received the
 * mousedown event or it may be some containing element. The second
 * argument must be the event object for the mousedown event.
 */
function beginDrag(elementToDrag, event) {
    // Figure out where the element currently is
    // The element must have left and top CSS properties in a style
attribute
    // Also, we assume they are set using pixel units
    var x = parseInt(elementToDrag.style.left);
```

```

var y = parseInt(elementToDrag.style.top);

// Compute the distance between that point and the mouse-click
// The nested moveHandler function below needs these values
var deltaX = event.clientX - x;
var deltaY = event.clientY - y;

// Register the event handlers that will respond to the mousemove
// and mouseup events that follow this mousedown event. Note that
// these are registered as capturing event handlers on the
document.
// These event handlers remain active while the mouse button
remains
// pressed and are removed when the button is released.
document.addEventListener("mousemove", moveHandler, true);
document.addEventListener("mouseup", upHandler, true);

// We've handled this event. Don't let anybody else see it.
event.stopPropagation( );
event.preventDefault( );

/**
 * This is the handler that captures mousemove events when an
element
 * is being dragged. It is responsible for moving the element.
 */
function moveHandler(event) {
    // Move the element to the current mouse position, adjusted as
    // necessary by the offset of the initial mouse-click
    elementToDrag.style.left = (event.clientX - deltaX) + "px";
    elementToDrag.style.top = (event.clientY - deltaY) + "px";

    // And don't let anyone else see this event
    event.stopPropagation( );
}

/**
 * This is the handler that captures the final mouseup event that
 * occurs at the end of a drag
 */
function upHandler(event) {
    // Unregister the capturing event handlers
    document.removeEventListener("mouseup", upHandler, true);
    document.removeEventListener("mousemove", moveHandler, true);
    // And don't let the event propagate any further
    event.stopPropagation( );
}
}

```

You can use `beginDrag()` in an HTML file like the following (which is a simplified version of [Example 19-2](#) with the addition of dragging):

```

<script src="Drag.js"></script> <!-- Include the Drag.js script -->
<!-- Define the element to be dragged -->
<div style="position:absolute; left:100px; top:100px;

```

```

        background-color: white; border: solid black;"/>
<!-- Define the "handle" to drag it with. Note the onmousedown
attribute. -->
<div style="background-color: gray; border-bottom: dotted black;
        padding: 3px; font-family: sans-serif; font-weight: bold;"
        onmousedown="beginDrag(this.parentNode, event);">
Drag Me <!-- The content of the "titlebar" -->
</div>
<!-- Content of the draggable element -->
<p>This is a test. Testing, testing, testing.<p>This is a test.<p>Test.
</div>

```

The key here is the `onmousedown` attribute of the inner `<div>` element. Although `beginDrag()` uses the DOM Level 2 event model, we register it here using the Level 0 model for convenience. As we'll discuss in the next section, the event models can be mixed, and when an event handler is specified as an HTML attribute, the event object is available using the `event` keyword. (This is not part of the DOM standard but is a convention of the Netscape 4 and IE event models, which are described later.)

Here's another simple example of using `beginDrag()`; it defines an image that the user can drag, but only if the **Shift** key is held down:

```

<script src="Drag.js"></script>


```

Note the differences between the `onmousedown` attribute here and the one in the previous example.

19.2.8 Mixing Event Models

So far, we've discussed the traditional Level 0 event model and the new standard DOM Level 2 model. For backward compatibility, browsers that support the Level 2 model will continue to support the Level 0 event model. This means that you can mix event models within a document, as we did in the HTML fragments used to demonstrate the element-dragging script in the previous section.

It is important to understand that web browsers that support the Level 2 event model always pass an event object to event handlers -- even handlers registered by setting an HTML attribute or a JavaScript property using the Level 0 model. When an event handler is defined as an HTML attribute, it is implicitly converted to a function that has an argument named `event`. This means that such an event handler can use the identifier `event` to refer to the event object.

The DOM standard never formalized the Level 0 event model. It does not even require properties like `onclick` for HTML elements that support an `onclick` attribute. However,

the standard recognizes that the Level 0 event model will remain in use and specifies that implementations that support the Level 0 model treat handlers registered with that model as if they were registered using `addEventListener()`. That is, if you assign a function `f` to the `onclick` property of a document element `e` (or set the corresponding HTML `onclick` attribute), it is equivalent to registering that function as follows:

```
e.addEventListener("click", f, false);
```

When `f()` is invoked, it is passed an event object as its argument, even though it was registered using the Level 0 model. Furthermore, if you change the value of the `onclick` property from function `f` to function `g`, it is equivalent to this code:

```
e.removeEventListener("click", f, false);  
e.addEventListener("click", g, false);
```

Note, however, that the Level 2 specification does not say whether an event handler registered by assigning to the `onclick` property can be removed by calling `removeEventListener()`. At the time of this writing, the Mozilla/Netscape implementation does not allow this.

19.2.9 Synthesizing Events

The DOM Level 2 standard includes an API for creating and dispatching synthetic events. This API allows events to be generated under program control rather than under user control. Although this is not a commonly needed feature, it can be useful, for example, to produce regression tests that subject a DHTML application to a known sequence of events and verify that the result is the same. It could also be used to implement a macro playback facility to automate commonly performed user-interface actions. On the other hand, the synthetic event API is not suitable for producing self-running demo programs: you can create a synthetic mousemove event and deliver it to the appropriate event handlers in your application, but this does not actually cause the mouse pointer to move across the screen!

Unfortunately, at the time of this writing, the synthetic event API is not supported by Netscape 6, nor by the current version of Mozilla.

To generate a synthetic event, you must complete three steps:

- Create an appropriate event object.
- Initialize the fields of the event object.
- Dispatch the event object to the desired document element.

To create an event object, call the `createEvent()` method of the Document object. This method takes a single argument, which is the name of the event module for which an

event object should be created. For example, to create an event object suitable for use with a click event, call the method as follows:

```
document.createEvent("HTMLEvents");
```

To create an event object suitable for use with any of the mouse event types, call it like this instead:

```
document.createEvent("MouseEvents");
```

Note that the argument to `createEvent()` is plural. This is counterintuitive, but it is the same string that you'd pass to the `hasFeature()` method to test whether a browser supports an event module.

After creating an event object, the next step is to initialize its properties. The properties of event objects are always read-only, however, so you cannot directly assign values to them. Instead, you must call a method to perform the initialization. Although the earlier descriptions of the `Event`, `MouseEvent`, and other event objects mentioned only properties, each object also defines a single method for initializing the properties of the event. This initialization method has a name that depends on the type of event object to be initialized and is passed as many arguments as there are properties to be set. Note that you can call an event initialization method only before dispatching a synthetic event: you cannot use these methods to modify the properties of an event object that is passed to an event handler.

Let's look at a couple of examples. As you know, click events are part of the `HTMLEvents` module and use event objects of type `Event`. These objects are initialized with an `initEvent()` method, as follows:

```
e.initEvent("click", "true", "true");
```

On the other hand, mousedown events are part of the `MouseEvents` module and use event objects of the `MouseEvent` type. These objects are initialized with an `initMouseEvent()` method that takes many more arguments:

```
e.initMouseEvent("mousedown", true, false, // Event properties
                window, 1, // UIEvent properties
                0, 0, 0, 0, // MouseEvent properties
                false, false, false, false,
                0, null);
```

Note that you pass only the event module name to `createEvent()`. The name of the actual event type is passed to the event initialization method. The DOM standard does not require that you use one of the predefined names. You may create events using any event

type name you choose, as long as it does not begin with a digit or with the prefix "DOM" (in uppercase, lowercase, or mixed case). If you initialize a synthetic event with a custom event type name, you must register event handlers with that event type name as well.

After creating and initializing an event object, you can dispatch it by passing it to the `dispatchEvent()` method of the appropriate document element. `dispatchEvent()` is defined by the `EventTarget` interface, so it is available as a method of any document node that supports the `addEventListener()` and `removeEventListener()` methods. The element to which you dispatch an event becomes the event target, and the event object goes through the usual sequence of event propagation. At each stage of event propagation, the event object you created is passed to any event handlers that were registered for the event type you specified when you initialized the event. Finally, when event propagation finishes, your call to `dispatchEvent()` returns. The return value is `false` if any of the event handlers called the `preventDefault()` method on your event object and is `true` otherwise.

19.3 The Internet Explorer Event Model

The event model supported by Internet Explorer 4, 5, 5.5, and 6 is an intermediate model, halfway between the original Level 0 model and the standard DOM Level 2 model. The IE event model includes an `Event` object that provides details about events that occur. Instead of being passed to event handler functions, however, the `Event` object is made available as a property of the `Window` object. The IE model supports event propagation by bubbling, but not by capturing, as the DOM model does. In IE 4, event handlers are registered in the same way as they are in the original Level 0 model. In IE 5 and later, however, multiple handlers may be registered with special (but nonstandard) registration functions.

The following sections provide more detail about this event model and document it by comparison to the original Level 0 event model and the standard Level 2 event model. Therefore, you should be sure you understand those two event models before reading about the IE model.

19.3.1 The IE Event Object

Like the standard DOM Level 2 event model, the IE event model provides details about each event that occurs in the properties of an `Event` object. The `Event` objects defined in the standard model were in fact modeled on the IE `Event` object, so you'll notice a number of similarities between the properties of the IE `Event` object and the properties of the `DOM Event`, `UIEvent`, and `MouseEvent` objects.

The most important properties of the IE `Event` object are:

`type`

A string that specifies the type of event that occurred. The value of this property is the name of the event handler with the leading "on" removed (e.g., "click" or "mouseover"). Compatible with the `type` property of the DOM Event object.

`srcElement`

The document element on which the event occurred. Comparable to the `target` property of the DOM Event object.

`button`

An integer that specifies the mouse button that was pressed. A value of 1 indicates the left button, 2 indicates the right button, and 4 indicates the middle button. If multiple buttons are pressed, these values are added together -- the left and right buttons together produce a value of 3, for example. Compare this with the `button` property of the DOM Level 2 MouseEvent object, but note that although the property names are the same, the interpretation of the property values differs.

`clientX, clientY`

These integer properties specify the mouse coordinates at the time of the event, relative to the upper-left corner of the containing window. Note that for documents that are larger than the window, these coordinates are not the same as the position within the document, and you may want to add the values `document.body.scrollLeft` and `document.body.scrollTop`, respectively, to account for scrolling. These properties are compatible with the DOM Level 2 MouseEvent properties of the same name.

`offsetX, offsetY`

These integer properties specify the position of the mouse pointer relative to the source element. They enable you to determine which pixel of an Image object was clicked on, for example. These properties have no equivalent in the DOM Level 2 event model.

`altKey, ctrlKey, shiftKey`

These boolean properties specify whether the **Alt**, **Ctrl**, and **Shift** keys were held down when the event occurred. These properties are compatible with the properties of the same name in the DOM Level 2 MouseEvent object. Note, however, that the IE Event object does not have a `metaKey` property.

`keyCode`

This integer property specifies the key code for keydown and keyup events and the Unicode character code for keypress events. Use `String.fromCharCode()` to convert character codes to strings. The DOM Level 2 event model does not standardize key events (although DOM Level 3 is working on this) and has no equivalent to these properties.

`fromElement, toElement`

`fromElement` specifies the document element that the mouse used to be over for mouseover events. `toElement` specifies the document element that the mouse has moved to for mouseout events. Comparable to the `relatedTarget` property of the DOM Level 2 MouseEvent object.

`cancelBubble`

A boolean property that, when set to `true`, prevents the current event from bubbling any further up the element containment hierarchy. Comparable to the `stopPropagation()` method of the DOM Level 2 Event object.

`returnValue`

A boolean property that can be set to `false` to prevent the browser from performing the default action associated with the event. This is an alternative to the traditional technique of returning `false` from the event handler. Comparable to the `preventDefault()` method of the DOM Level 2 Event object.

You can find complete documentation for the IE Event object in the client-side reference section of this book.

19.3.2 The IE Event Object as a Global Variable

Although the IE event model provides event details in an Event object, it never passes Event objects as arguments to event handlers. Instead, it makes the Event object available as the `event` property of the global Window object. This means that an event handling function in IE can refer to the Event object as `window.event` or simply as `event`. Although it seems strange to use a global variable where a function argument would do, the IE scheme works because it is implicit in the event-driven programming model that only one event is ever being processed at a time. Since two events are never handled concurrently, it is safe to use a global variable to store details on the event that is currently being processed.

The fact that the Event object is a global variable is incompatible with the standard DOM Level 2 event model, but there is a one-line workaround. If you want to write an event handler function that works with either event model, write the function so that it expects an argument, and then, if no argument is passed, initialize the argument from the global variable. For example:

```
function portableEventHandler(e) {
    if (!e) e = window.event; // Get event details for IE

    // Body of the event handler goes here
}
```

19.3.3 Event Bubbling in IE

The IE event model does not have any notion of event capturing, as the DOM Level 2 model does. However, events do bubble up through the containment hierarchy in the IE model, just as they do in the Level 2 model. As with the Level 2 model, event bubbling applies only to raw or input events (primarily mouse and keyboard events), not to higher-level semantic events. The primary difference between event bubbling in the IE and DOM Level 2 event models is the way that you stop bubbling. The IE Event object does not have a `stopPropagation()` method, as the DOM Event object does. To prevent an event from bubbling or stop it from bubbling any further up the containment hierarchy, an IE event handler must set the `cancelBubble` property of the Event object to `true`:

```
window.event.cancelBubble = true;
```

Note that setting `cancelBubble` applies only to the current event. When a new event is generated, a new Event object is assigned to `window.event`, and `cancelBubble` is restored to its default value of `false`.

19.3.4 IE Event-Handler Registration

In IE 4, event handlers are registered in the same way they are in the original Level 0 event model: by specifying them as HTML attributes or assigning functions to the event handler properties of document elements. The only difference is that IE 4 allows access to (and event-handler registration on) all of the elements in a document, instead of just the form, image, and link elements that are accessible with the Level 0 DOM.

IE 5 and later introduce the `attachEvent()` and `detachEvent()` methods, which provide a way to register more than one handler function for a given event type on a given object. These methods work like `addEventListener()` and `removeEventListener()`, except that since the IE event model does not support event capturing, they expect only two arguments: the event type and the handler function. Also, unlike with the Level 2 event model, the event handler names passed to the IE method

should include the "on" prefix: use "onclick" instead of just "click". You can use `attachEvent()` to register an event handler as follows:

```
function highlight( ) { /* Event-handler code goes here */ }
document.getElementById("myelt").attachEvent("onmouseover", highlight);
```

Another difference between `attachEvent()` and `addEventListener()` is that functions registered with `attachEvent()` are invoked as global functions, rather than as methods of the document element on which the event occurred. That is, when an event handler registered with `attachEvent()` executes, the `this` keyword refers to the Window object, not to the event's target element.

19.3.5 Example: Dragging with the IE Event Model

[Example 19-3](#) is a modified version of the `beginDrag()` function that was presented in [Example 19-2](#). This version includes code that makes it work with the IE event model, in addition to the DOM Level 2 event model. The design and intended usage of this version of `beginDrag()` are the same as in [Example 19-2](#), so if you understood that example, you should have no trouble understanding this one. What makes this example interesting is that it juxtaposes two event models, clearly highlighting their differences.

The biggest difference in the IE version of the code is that it must rely on event bubbling rather than event capturing. This usually works, but it is not the ideal solution for this problem. Another important difference to note is that IE event handlers are not passed an Event object. Note that the code in this example also distinguishes between IE 5 and later, which support `attachEvent()`, and IE 4, which does not. See the discussion of [Example 19-2](#) for a sample HTML document that is designed to use this `beginDrag()` function.

Example 19-3. Dragging with the IE event model

```
/**
 * PortableDrag.js:
 * beginDrag( ) is designed to be called from an onmousedown event
handler.
 * elementToDrag may be the element that received the mousedown event,
or it
 * may be some containing element. event must be the Event object for
the
 * mousedown event. This implementation works with both the DOM Level 2
 * event model and the IE event model.
 */
function beginDrag(elementToDrag, event) {
    // Compute the distance between the upper-left corner of the
element
    // and the mouse-click. The moveHandler function below needs these
values.
    var deltaX = event.clientX - parseInt(elementToDrag.style.left);
    var deltaY = event.clientY - parseInt(elementToDrag.style.top);
```

```

    // Register the event handlers that will respond to the mousemove
events
    // and the mouseup event that follow this mousedown event.
    if (document.addEventListener) { // DOM Level 2 Event Model
        // Register capturing event handlers
        document.addEventListener("mousemove", moveHandler, true);
        document.addEventListener("mouseup", upHandler, true);
    }
    else if (document.attachEvent) { // IE 5+ Event Model
        // In the IE event model, we can't capture events, so these
handlers
        // are triggered only if the event bubbles up to them.
        // This assumes that there aren't any intervening elements that
        // handle the events and stop them from bubbling.
        document.attachEvent("onmousemove", moveHandler);
        document.attachEvent("onmouseup", upHandler);
    }
    else { // IE 4 Event Model
        // In IE 4 we can't use attachEvent( ), so assign the event
handlers
        // directly after storing any previously assigned handlers, so
they
        // can be restored. Note that this also relies on event
bubbling.
        var oldmovehandler = document.onmousemove;
        var olduphandler = document.onmouseup;
        document.onmousemove = moveHandler;
        document.onmouseup = upHandler;
    }

    // We've handled this event. Don't let anybody else see it.
    if (event.stopPropagation) event.stopPropagation( ); // DOM Level
2
    else event.cancelBubble = true; // IE

    // Now prevent any default action.
    if (event.preventDefault) event.preventDefault( ); // DOM Level
2
    else event.returnValue = false; // IE

/**
 * This is the handler that captures mousemove events when an
element
 * is being dragged. It is responsible for moving the element.
 */
function moveHandler(e) {
    if (!e) e = window.event; // IE Event Model

    // Move the element to the current mouse position, adjusted as
    // necessary by the offset of the initial mouse-click.
    elementToDrag.style.left = (e.clientX - deltaX) + "px";
    elementToDrag.style.top = (e.clientY - deltaY) + "px";

    // And don't let anyone else see this event.
    if (e.stopPropagation) e.stopPropagation( ); // DOM Level 2
    else e.cancelBubble = true; // IE
}

```

```

/**
 * This is the handler that captures the final mouseup event that
 * occurs at the end of a drag.
 */
function upHandler(e) {
    if (!e) e = window.event; // IE Event Model

    // Unregister the capturing event handlers.
    if (document.removeEventListener) { // DOM Event Model
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler,
true);
    }
    else if (document.detachEvent) { // IE 5+ Event Model
        document.detachEvent("onmouseup", upHandler);
        document.detachEvent("onmousemove", moveHandler);
    }
    else { // IE 4 Event Model
        document.onmouseup = olduphandler;
        document.onmousemove = oldmovehandler;
    }

    // And don't let the event propagate any further.
    if (e.stopPropagation) e.stopPropagation( ); // DOM Level 2
    else e.cancelBubble = true; // IE
}
}

```

19.4 The Netscape 4 Event Model

The Netscape 4 event model is like the original Level 0 event model, except that it provides event details in an Event object that is passed as an argument to handler functions. It also supports special methods to enable event capturing. These features are explained in the sections that follow.

19.4.1 The Netscape 4 Event Object

The Netscape 4 event model defines an Event object that contains details about the event that occurred. Like the DOM Level 2 model, it passes an Event object as an argument to all event handlers. Unfortunately, however, the properties of the Netscape 4 Event object are almost entirely different than those of the IE Event object and the various DOM Level 2 event objects. The key Event properties in the Netscape 4 event model are:

type

A string that specifies the type of event that occurred. This string is the name of the event handler, minus the "on" prefix (e.g., "click" or "mousedown"). This property is compatible with the IE and DOM Level 2 Event objects.

`target`

The document element on which the event occurred. This property is compatible with the `target` property of the DOM Level 2 Event object and comparable to `srcElement` in the IE Event object.

`pageX`, `pageY`

These properties specify the pixel coordinates at which the event occurred, relative to the upper-left corner of the window. For documents that are larger than the window, you need to add in the offsets `window.pageXOffset` and `window.pageYOffset` to convert these to document coordinates. Comparable to the `clientX` and `clientY` properties of the DOM Level 2 MouseEvent object and the IE Event object.

`which`

An integer that specifies which mouse button or key was pressed. For mouse events, the left, middle, and right buttons are specified by the values 1, 2, and 3, respectively. Compare this to the (mutually incompatible) `button` properties of the DOM Level 2 MouseEvent object and the IE Event object. For keyboard events, this property contains the Unicode encoding of the key that was pressed. Compare this to the `keyCode` property of the IE Event object.

`modifiers`

An integer that specifies which keyboard modifier keys were pressed when the event occurred. The value is a bitmask comprised of any of the following values: `Event.ALT_MASK`, `Event.CONTROL_MASK`, `Event.META_MASK`, and `Event.SHIFT_MASK`. Comparable to the `altKey`, `ctrlKey`, `metaKey`, and `shiftKey` properties of the DOM Level 2 MouseEvent object and the IE Event object.

In the Netscape 4 event model, an Event object is passed to all event handlers. When an event handler is defined as a string of JavaScript code in an HTML attribute, that code is implicitly converted to a function with an argument named `event`. This means that HTML event handlers can refer to the Event object with the identifier `event`. (Compare

this to the IE model, in which the `event` identifier refers to the global Event object. The implementations are quite different, but the practical result is the same.)

For backward compatibility, the Event objects used by Mozilla and Netscape 6 implement most of the properties of the Netscape 4 Event object, with the notable exception, at the time of this writing, of the `modifiers` property.

19.4.2 Event Capturing in Netscape 4

The Netscape 4 event model does not support event bubbling, as the IE event model does, but it does support a limited form of event capturing, like the DOM Level 2 model does. (In fact, the event-propagation model for the DOM standard is a combination of the Netscape capturing and IE bubbling models.) Although Netscape 4 supports a form of event capturing, the way it works is quite different from that defined by the DOM Level 2 event model.

In Netscape 4, the Window, Document, and Layer objects may request the opportunity to preview certain types of events before they are processed by the elements that generated them. Such a request is made with the `captureEvents()` method of these objects. The argument to this method specifies the type of events to be captured; it is a bitmask composed of constants defined as static properties of the Event constructor. So, for example, if a program wants all mousedown and mouseup events to be routed to the Window object before being handled by the object for which they were intended, it can call `captureEvents()` like this:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);
```

Having made this request to receive the events, the program then has to register event handlers for those events:

```
window.onmousedown = function(event) { ... };  
window.onmouseup = function(event) { ... };
```

When one of these capturing event handlers receives an event, it gets to decide what should happen to it next. In some programs, a captured event is handled and propagates no further. In other circumstances, however, the program wants to pass the event along. If you pass the event to the `routeEvent()` method of the Window, Document, and Layer objects, the method passes the event to the next Window, Document, or Layer object that has used `captureEvents()` to specify interest in that type of event. Or, if there is no other capturing object to which to route the event, it is routed to its original source object and the appropriate event handler of that object is invoked. For example:

```
function clickHandler(event) {  
    if (event.which == 3) { // It is the right mouse button  
        // Handle the event here, and do nothing else
```

```

    // The event will not propagate any further
  }
  else { // It is not the right mouse button
    // We're not interested in this event, so let it propagate on
    // to some element that is interested in it
    window.routeEvent(event);
  }
}
}

```

An alternative to calling `routeEvent()` is to simply pass the Event object to the `handleEvent()` method of the object to which you want the event delivered. The `handleEvent()` method passes the event to the appropriate event handler of that object.

When a Window, Document, or Layer object no longer wishes to capture events, it should call the `releaseEvents()` method, specifying the same argument it passed to `captureEvents()`.

The Netscape 4 event-capturing model is fundamentally incompatible with the DOM Level 2 event-capturing model. For example, the DOM model propagates captured events by default, but the Netscape model does not. Mozilla and Netscape 6 implement the Netscape 4 event-capturing API, but the API appears to be nonfunctional.

19.4.3 Example: Dragging with the Netscape 4 Event Model

[Example 19-4](#) is an implementation of our familiar `beginDrag()` method, using the Netscape 4 event model (and the Netscape 4 Layer-based DOM). It demonstrates how events are captured and how event handlers are written for this event model. This example includes both JavaScript code and a simple HTML document that uses the `beginDrag()` method to define an image that the user can drag. Compare this implementation of `beginDrag()` to the two we've seen previously. Note that this example defines its nested event handler functions at the beginning of the `beginDrag()` function instead of at the end. This is a bug workaround: if the nested functions are placed at the end of `beginDrag()`, they do not work in Netscape 4. Also note the `onmousedown` handler at the end of the example: it allows dragging only if the **Shift** key is held down and tests for this modifier key using the Netscape 4 Event object API, which is significantly different from the DOM Level 2 and IE APIs.^[2]

^[2] At the time of this writing, Mozilla and Netscape 6 have not retained compatibility with the `modifiers` property of the Netscape 4 Event object, so the `onmousedown` handler shown here works only in Netscape 4, not in Netscape 6.

Example 19-4. Dragging in Netscape 4

```

<script>
/**
 * This function is intended for use in a mousedown event handler of an
 * object
 * within a layer. The first argument must be a Layer object. The
 * second
 * argument must be the Event object for the mousedown event.

```

```

    **/
function beginDrag(layerToDrag, event) {
    // This nested function responds to mousemove events and moves the
    layer
    function moveHandler(event) {
        // Move the element to the current mouse position, adjusted as
        // necessary by the offset of the initial mouse-click
        layerToDrag.moveTo(event.pageX - deltaX, event.pageY-deltaY);

        // Don't take any default action, and don't propagate further
        return false;
    }

    // This nested function handles mouseup events
    // It stops capturing events and deregisters the handlers
    function upHandler(event) {
        // Stop capturing and handling drag events
        document.releaseEvents(Event.MOUSEMOVE | Event.MOUSEUP);
        document.onmousemove = null;
        document.onmouseup = null;

        // Don't take any default action, and don't propagate further
        return false;
    }

    // Compute the distance between the upper-left corner of the layer
and
    // the mouse-click. The moveHandler function below needs these
values.
    var deltaX = event.pageX - layerToDrag.left;
    var deltaY = event.pageY - layerToDrag.top;

    // Arrange to capture mousemove and mouseup events
    // Then arrange to handle them using the functions defined below
    document.captureEvents(Event.MOUSEMOVE | Event.MOUSEUP);
    document.onmousemove = moveHandler;
    document.onmouseup = upHandler;
}
</script>
<!-- Here's how we might use beginDrag( ) in Netscape 4 -->
<!-- Define a layer using CSS attributes -->
<div id="div1" style="position:absolute; left:100px; top:100px;">
<!-- Give the layer some content and a mousedown event handler -->

</div>

```

Chapter 20. Compatibility Techniques

JavaScript, like Java, is one of a new breed of platform-independent languages. That is, you can develop a program in JavaScript and expect to run it unchanged in a JavaScript-enabled web browser running on any type of computer with any type of operating system. Though this is the ideal, we live in an imperfect world and have not yet reached that state of perfection.

There are, and probably always will be, compatibility problems that JavaScript programmers must bear in mind. The one fact that we must always remember is that it is a heterogeneous network out there. Your JavaScript programs may run on three or more operating systems, using three or more versions of browsers from at least two different vendors. This can be difficult to keep in mind for those of us who come from the nonportable past, when programs were developed on a platform-specific basis. Remember: which platform you develop a program on doesn't matter. It may work fine on that platform, but the real test is whether it works (or fails gracefully) on *all* platforms on which it is used.

The compatibility issues fall into two broad categories: platform-specific, browser-specific, and version-specific features on one hand; and bugs and language-level incompatibilities, including the incompatibility of JavaScript with non-JavaScript browsers, on the other. This chapter discusses techniques for coping with compatibility issues in both of these areas. If you've worked your way through all the previous chapters in this book, you are probably an expert JavaScript programmer, and you may already be writing serious JavaScript programs. Don't release those programs on the Internet (or onto a heterogeneous intranet) before you've read this chapter, though!

20.1 Platform and Browser Compatibility

When developing production-quality JavaScript code, testing and knowledge of platform-specific, vendor-specific, and version-specific incompatibilities are your chief allies. If you know, for example, that Netscape 2 on Macintosh platforms always gets the time wrong by about an hour, you can take steps to deal with this problem. If you know that Netscape 2 and 3 on Windows platforms do not automatically clear your setting of the status line when the mouse moves off a hypertext link, you can provide an appropriate event handler to explicitly clear the status line. If you know that Internet Explorer 4 and Netscape 4 support vastly different Dynamic HTML models, you can write pages that use the appropriate mechanism depending on the browser in use.

Knowledge of existing incompatibilities is crucial to writing compatible code. Unfortunately, producing a definitive listing of all known vendor, version, and platform incompatibilities would be an enormous task. It is beyond the scope and mission of this book, and it has apparently never even been seriously attempted. You may find some assistance on the Internet, but you will have to rely primarily on your own experience and

testing. Once you have identified an area of incompatibility, however, there are a number of basic approaches you can take to coping with it, as described in the following sections.

20.1.1 The Least-Common-Denominator Approach

One technique for dealing with incompatibilities is to avoid them like the plague. For example, the Date object is notoriously buggy in Netscape 2. If you want Netscape 2 users to be able to use your programs, you can simply avoid relying on the Date object at all.^[1]

^[1] I don't actually recommend doing this. At the time of this writing, Netscape 2 is so far out of date that it is safe to ignore it.

As another example, Netscape 3 and IE 3 both support the `opener` property of the Window object, but Netscape 2 does not. The least-common-denominator approach says that you should not use this property if compatibility with Netscape 2 is a goal. Instead, you can create an equivalent property of your own whenever you open a new window:

```
newwin = window.open("", "new", "width=500, height=300");
newwin.creator = self;
```

If you consistently set a `creator` property for each new window you create, you can rely on that property instead of the nonportable `opener` property. (Another alternative, as we'll see later, is to give up on compatibility with Netscape 2 and require a browser that supports JavaScript 1.1 or later, as all such browsers support the `opener` property.)

With this technique, you use only features that are known to work on all your target platforms. It doesn't allow you to write cutting-edge programs or push the envelope, but it results in portable, safe programs that can serve many important functions.

20.1.2 Defensive Coding

With the defensive coding approach to compatibility, you write code that contains platform-independent workarounds for platform-specific incompatibilities. For example, if you set the `status` property of a Window object from the `onmouseover` event handler to display a custom message in the status line, the status line is cleared when you move the mouse off the hyperlink, except in Windows versions of Netscape 2 and 3. To correct for this problem, you could get in the habit of including an `onmouseout` event handler to clear the status line. This precaution fixes the bug in current (and future) platforms that have it and doesn't do any harm on platforms that don't have the bug.

20.1.3 Feature Testing

Feature testing is a powerful technique for coping with incompatibilities. If you want to use a feature that may not be supported by all browsers, include code in your script that tests to see whether that feature is supported. If the desired feature is not supported on the

current platform, either do not use it on that platform or provide alternative code that works on all platforms.

Consider again the `opener` property. In the least-common-denominator approach, we simply avoided the use of this property and used an alternative on all platforms. With the feature-testing approach, we provide the alternative only when the current platform does not support `opener`:

```
newwin = window.open("", "new", "width=500, height=300");
if (!newwin.opener) newwin.opener = self;
```

Note how we tested for the existence of the `opener` property. The same technique works to test for the existence of methods. For example, the `split()` method of the String object exists only for JavaScript 1.1 implementations. We can write our own version of this function that works in all versions of JavaScript, but for efficiency we'd like to use the fast, built-in method on those platforms that do support it. Thus, our feature-testing code to `split()` a string might end up looking like this:

```
if (s.split)           // Check if the method exists, without
    invoking it
    a = s.split(":");   // If it does exist, it is safe to invoke it
else                   // Otherwise:
    a = mysplit(s, ":"); // use our alternative implementation
```

Feature testing is commonly used for performing DHTML effects that are supported only on some browsers or are implemented differently in different browsers. For example, if you are designing a site that includes image rollover effects, you can use feature testing with code like this:

```
if (document.images) { // If the browser defines an images[] array,
    // we include image rollover code here
}
// Otherwise, we simply omit the image rollover effect
```

As another example, suppose we want to work with a dynamically positioned document element. Different browsers have different APIs for doing this, so we first use feature testing to see which API is supported by the current browser with code like this:

```
if (document.getElementById) { // If the W3C DOM API is supported,
    // do our DHTML using the W3C DOM API
}
else if (document.all) {      // If the IE 4 API is supported,
    // do our DHTML using the IE 4 API
}
else if (document.layers) {   // If the Netscape 4 API is supported,
    // do the DHTML effect (as best we can) using the Netscape 4 API
}
```

```
else { // Otherwise, DHTML is not supported,  
    // so provide a static alternative to DHTML, if we can  
}
```

The nice thing about the feature-testing technique is that it results in code that is not tied to a specific list of browser vendors or browser version numbers. It works with the set of browsers that exist today and should continue to work with future browsers, whatever feature sets they implement.

20.1.4 Platform-Specific Workarounds

Feature testing is well suited to checking for support of large functional areas. You can use it to determine whether a browser supports image rollovers or the W3C DOM API, for example. On the other hand, sometimes you may need to work around individual bugs or quirks in a particular browser, and there may be no easy way to test for the existence of the bug. In this case, you will need to create a platform-specific workaround that is tied to a particular browser vendor, version, or operating system (or some combination of the three).

Recall from [Chapter 13](#) that the `navigator` property of the Window object provides information about the vendor and version of the browser and the operating system on which it is running. You can use this information to insert platform-specific code into your program.

An example of a platform-specific workaround involves the `bgColor` property of the Document object. On Windows and Macintosh platforms, you can set this property at runtime to change the background color of a document. Unfortunately, when you do this on Unix versions of Netscape 2 and 3, the color changes but the document contents temporarily disappear. If you wanted to create a special effect using a changing background color, you could use the Netscape object to test for Unix platforms and simply skip the special effect for those platforms. The code could look like this:

```
// Check whether we're running Netscape 2 or 3 on a Unix platform  
var nobg = (parseInt(navigator.appVersion) < 4) && // Version  
           (navigator.appName.indexOf("Netscape") != -1) && // Vendor  
           (navigator.appVersion.indexOf("X11") != -1); // OS  
  
// If we're not, then go ahead and animate the page background color  
if (!nobg) animate_bg_color( );
```

When writing platform-specific workarounds, it is common to use "client-sniffer" code to determine what the current platform is, based (typically) on the properties of the `navigator` object. You run your client-sniffer code once, and it sets variables that describe the current platform. Then you don't have to reparse the properties of `navigator` for each platform-specific bit of code you write; you can simply use the variables set by

the sniffer code. A simple sniffer that may be sufficient for many purposes might look like this:

```
var browserVersion = parseInt(navigator.appVersion);
var isNetscape = navigator.appName.indexOf("Netscape") != -1;
var isIE = navigator.appName.indexOf("Microsoft") != -1;
var agent = navigator.userAgent.toLowerCase( );
var isWindows = agent.indexOf("win") != -1;
var isMac = agent.indexOf("mac") != -1;
var isUnix = agent.indexOf("X11") != -1;
```

With variables like these defined, you might write code like the following:

```
if (isNetscape && browserVersion < 4 && isUnix) {
    // Work around a bug in Netscape 3 on Unix platforms here
}
```

A variety of prewritten client sniffers are available on the Internet. You can find a thorough one (along with a helpful discussion of its use) at http://www.mozilla.org/docs/web-developer/sniffer/browser_type.html.

20.1.5 Compatibility Through Server-Side Scripts

Another platform-specific approach to compatibility is possible if your web application includes the use of server-side scripts, such as CGI scripts or server-side JavaScript. A program on the server side can inspect the `User-Agent` field of the HTTP request header, which allows it to determine exactly what browser the user is running. With this information, the program can generate customized JavaScript code that is known to work correctly on that browser. Or, if the server-side script detects that the user's browser does not support JavaScript, it can generate web pages that do not require JavaScript at all. An important drawback to this approach is that a server-side script cannot detect when a user has disabled JavaScript support in her browser.

Note that the topics of CGI programming and server-side scripting in general are beyond the scope of this book.

20.1.6 Ignore the Problem

An important question to ask when considering any incompatibility is, how important is it? If the incompatibility is minor or cosmetic, affects a browser or platform that is not widely used, or affects only an out-of-date version of a browser, you might simply decide to ignore the problem and let the users affected by it cope with it on their own.

For example, earlier I suggested defining an `onmouseout` event handler to correct for the fact that Netscape 2 and 3 for Windows do not correctly clear the status line.

Unfortunately, the `onmouseout` event handler is not supported in Netscape 2, so this

workaround won't work for that platform. If you expect your application to have a lot of users who use Netscape 2 on Windows and you think that it is really important to get that status line cleared, you'll have to develop some other workaround. You could use `setTimeout()` in your `onmouseover` event handler to arrange for the status line to be cleared in two seconds. But this solution brings problems with it: what if the mouse is still over the hypertext link and the status line shouldn't be cleared in two seconds? In this case, a simpler approach might be to simply ignore the problem. This approach can easily be justified, because Netscape 2 is by now well out of date; any users still relying on it should be encouraged to upgrade.

20.1.7 Fail Gracefully

Finally, there are some incompatibilities that cannot be ignored and cannot be worked around. In these cases, your program should work correctly on all platforms, browsers, and versions that provide the needed features and fail gracefully on all others. Failing gracefully means recognizing that the required features are not available and informing the user that he will not be able to use your JavaScript program.

For example, the image-replacement technique we saw during the discussion of images in [Chapter 14](#) does not work in Netscape 2 or Internet Explorer 3, and there is really no workaround that can simulate it. Therefore, we should not even attempt to run the program on those platforms; instead, we should politely notify the user of the incompatibility.

Failing gracefully can be harder than it sounds. Much of the rest of this chapter explains techniques for doing so.

20.2 Language Version Compatibility

The previous section discussed general compatibility techniques that are useful for coping with incompatibilities between different versions of browsers from different vendors running on different platforms. This section addresses another compatibility concern: how to use new features of the JavaScript language in a way that does not cause errors on browsers that do not support those features. Our goals are simple: we need to prevent JavaScript code from being interpreted by browsers that don't understand it, and we need to display special messages on those browsers that inform users that their browsers cannot run the scripts.

20.2.1 The language Attribute

The first goal is easy. As we saw in [Chapter 12](#), we can prevent a browser from attempting to run code that it cannot understand by setting the `language` attribute of the `<script>` tag appropriately. For example, the following `<script>` tag specifies that the code it contains uses features of JavaScript 1.1 and that browsers that do not support that version of the scripting language should not attempt to run it:

```
<script language="JavaScript1.1">
  // JavaScript 1.1 code goes here
</script>
```

Note that the use of the `language` attribute is a general technique. When set to the string "JavaScript1.2", the attribute prevents JavaScript 1.0 or 1.1 browsers from attempting to run the code. At the time of this writing, the latest browsers (Netscape 6 and IE 6) support language versions 1.0, 1.1, 1.2, 1.3, 1.4, and 1.5. If you write JavaScript code that includes the `try/catch` exception-handling statement, for example, you should include it in a `<script>` tag with `language="JavaScript1.5"` to prevent browsers that do not understand this statement from trying to run it.

Unfortunately, the `language` attribute is marred by the fact that specifying `language="JavaScript1.2"` causes Netscape to behave in ways that are incompatible with the ECMA-262 standard. For example, as we saw in [Chapter 5](#), setting the `language` attribute to this value causes the `==` operator to perform equality comparisons without doing any type conversions. And as we saw in [Chapter 8](#), specifying "JavaScript1.2" also causes the `toString()` method to behave quite differently. Unless you explicitly want these new, incompatible behaviors, or unless you can carefully avoid all incompatible features, you should avoid the use of `language="JavaScript1.2"`.

Note that the version numbers used by the `language` attribute match the version numbers of Netscape's (and now Mozilla's) JavaScript interpreter. Microsoft's interpreter has more or less followed the evolution of Netscape's, but bear in mind that the `language` attribute is still somewhat vendor-specific: the language features supported by different vendors for a given version number are not guaranteed to be the same. This is particularly so for `language="JavaScript1.2"`, but caution is advisable for other versions as well. Unfortunately, there is no way to specify a specification version with the `language` attribute. That is, you *cannot* write:

```
<script language="ECMAScript3">...</script>
```

20.2.2 Explicit Version Testing

The `language` attribute provides at least a partial solution to the problem of language version compatibility, but it solves only half of the problem. We also need to be able to fail gracefully for browsers that do not support the desired version of JavaScript. If we require JavaScript 1.1, we'd like to be able to notify users of JavaScript 1.0 browsers that they cannot use the page. [Example 20-1](#) shows how we can do this.

Example 20-1. A message for browsers that do not support JavaScript 1.1

```
<!-- Set a variable to determine what version of JavaScript we support
-->
<!-- This technique can be extended to any number of language versions
-->
```

```

<script language="JavaScript"> var _version = 1.0; </script>
<script language="JavaScript1.1"> _version = 1.1; </script>
<script language="JavaScript1.2"> _version = 1.2; </script>

<!-- Run this code on any JavaScript-enabled browser -->
<!-- If the version is not high enough, display a message -->
<script language="JavaScript">
  if (_version < 1.1) {
    document.write('<hr><h1>This Page Requires JavaScript 1.1</h1>');
    document.write('Your JavaScript 1.0 browser cannot run this
page.<hr>');
  }
</script>

<!-- Now run the actual program only on JavaScript 1.1 browsers -->
<script language="JavaScript1.1">
  // The actual JavaScript 1.1 code goes here
</script>

```

20.2.3 Suppressing Version-Related Errors

[Example 20-1](#) showed how we can write JavaScript 1.1 code that JavaScript 1.0 browsers do not attempt to execute. What if we wanted to write JavaScript 1.2 code that JavaScript 1.1 browsers do not attempt to execute? We could use the `language` attribute to explicitly specify "JavaScript1.2", but as we discussed earlier, this causes Netscape to behave incompatibly. Unfortunately, JavaScript 1.2 adds a lot of new syntax to the language. If you write code that uses a `switch` statement, an object initializer, or a function literal and then run that code on a JavaScript 1.1 browser, you'll cause runtime syntax errors.

One way to work around this problem is simply to suppress any errors that occur on JavaScript 1.1 browsers. [Example 20-2](#) shows how this can be done using the `onerror` error handler of the Window object (which was described in [Chapter 13](#)).

Example 20-2. Suppressing version-related errors

```

<!-- Check whether JavaScript 1.2 is supported -->
<script language="JavaScript1.2">var _js12_ = 1.2</script>

<!-- Now avoid the problems with JavaScript 1.2 on Netscape by running
-->
<!-- the following code on any browser that supports JavaScript 1.1. If
-->
<!-- the browser does not support JavaScript 1.2, however, we'll
display -->
<!-- an error message and suppress any syntax errors that occur. -->
<script language="JavaScript1.1">
// If JavaScript 1.2 is not supported, fail gracefully
function suppressErrors( ) { return true; }
if (!_js12_) {
  window.onerror = suppressErrors;
  alert("This program requires a browser with JavaScript 1.2
support");
}

```

```
// Now proceed with the JavaScript 1.2 code
</script>
```

20.2.4 Loading a New Page for Compatibility

Another approach to version compatibility is to load a web page that requires a specific level of JavaScript support only after determining whether the browser provides that level of support. [Example 20-3](#) shows how this might be done with a short script that tests whether JavaScript 1.2 is supported. If the browser supports this version, the script uses the `Location.replace()` method to load in a new web page that requires JavaScript 1.2. If JavaScript 1.2 is not supported, the script displays a message saying that it is required.

Example 20-3. A web page to test for JavaScript compatibility

```
<head>
<script language="JavaScript1.2">
// If JavaScript 1.2 is supported, extract a new URL from the portion
of
// our URL following the question mark, and load in that new URL
location.replace(location.search.substring(1));

// Enter a really long, empty loop, so that the body of this document
// doesn't get displayed while the new document is loading
for(var i = 0; i < 10000000; i++);
</script>
</head>
<body>
<hr size="4">
<h1>This Page Requires JavaScript 1.2</h1>
Your browser cannot run this page. Please upgrade to a browser that
supports JavaScript 1.2, such as Netscape 4 or Internet Explorer 4.
<hr size="4">
</body>
```

The most interesting thing about this example is that it is a generic one -- the name of the JavaScript 1.2 file to be loaded is encoded in the search portion of the original URL; that file is loaded only if JavaScript 1.2 is supported. Thus, if the file in this example has the name *testjs12.html*, you can use it in URLs like the one shown in this hyperlink:

```
<a
href="http://my.isp.net/~david/utils/testjs12.html?../js/cooljs12.html"
>
Visit my cool JavaScript 1.2 page!
</a>
```

The other thing to note about [Example 20-3](#) is that calling `Location.replace()` starts a new page loading but does not immediately stop the current page from loading.

Therefore, the JavaScript code in this example enters a long, empty loop after it calls `replace()`. This prevents the rest of the document from being parsed and displayed, so that users of JavaScript 1.2 browsers do not see the message intended for users of browsers that do not support JavaScript 1.2.

Finally, note that the technique shown in [Example 20-3](#) is useful not only to distinguish one version of JavaScript from another, but also to distinguish between browsers that support JavaScript and those that do not. The next section discusses other compatibility techniques that are useful with non-JavaScript browsers.

20.3 Compatibility with Non-JavaScript Browsers

The previous section discussed compatibility with browsers that do not support a particular version of JavaScript. This section considers compatibility with browsers that do not support JavaScript at all. These are either browsers that have no JavaScript capability or browsers in which the user has disabled JavaScript (which some users do because of security concerns). Because a number of such browsers are still in use, you should design your web pages to fail gracefully when read into browsers that do not understand JavaScript. There are two parts to doing this: first, you must take care to ensure that your JavaScript code does not appear as if it were HTML text; and second, you should arrange to display a message informing the visitor that her browser cannot correctly handle the page.

20.3.1 Hiding Scripts from Old Browsers

Web browsers that support JavaScript execute the JavaScript statements that appear between the `<script>` and `</script>` tags. Browsers that don't support JavaScript but recognize the `<script>` tag simply ignore everything between `<script>` and `</script>`. This is as it should be. Really old browsers, however (and there are still some out there), do not even recognize the `<script>` and `</script>` tags. This means that they ignore the tags themselves and treat all the JavaScript between them as HTML text to be displayed. Unless you take steps to prevent it, users of these old browsers see your JavaScript code formatted into big meaningless paragraphs and presented as web page content!

To prevent this, enclose the body of your script within an HTML comment, using the format shown in [Example 20-4](#).

Example 20-4. A script hidden from old browsers

```
<script language="JavaScript">
<!-- Begin HTML comment that hides the script
    // JavaScript statements go here
    //
    //
// End HTML comment that hides the script -->
</script>
```

Browsers that do not understand the `<script>` and `</script>` tags simply ignore them. Thus, lines one and seven in [Example 20-4](#) have no effect on these browsers. They'll ignore lines two through six as well, because the first four characters on line two begin an HTML comment and the last three characters on line six end that comment -- everything in between is ignored by the HTML parser.

This script-hiding technique also works for browsers that *do* support JavaScript. Lines one and seven indicate the beginning and end of a script. Client-side JavaScript interpreters recognize the HTML comment-opening string `<!--` but treat it as a single-line comment. Thus, a browser with JavaScript support treats line two as a single-line comment. Similarly, line six begins with the `//` single-line comment string, so that line is ignored by JavaScript-enabled browsers as well. This leaves lines three through five, which are executed as JavaScript statements.

While it takes a little getting used to, this simple and elegant mix of HTML and JavaScript comments does exactly what we need: it prevents JavaScript code from being displayed by browsers that do not support JavaScript. Although a declining number of browsers require this type of commenting, it is still quite common to see it used in JavaScript code on the Internet. The comments need not be as verbose as in [Example 20-4](#), of course. It is common to see scripts like this:

```
<script language="JavaScript">
<!--
    document.write(new Date(  ));
// -->
</script>
```

This commenting technique has solved the problem of hiding our JavaScript code from browsers that can't run it. The next step in failing gracefully is to display a message to the user to let him know that the page cannot run.

20.3.2 `<noscript>`

The `<noscript>` and `</noscript>` tags enclose an arbitrary block of HTML text that should be displayed by any browser that does not support JavaScript. These tags can be employed to let a user know that his browser cannot correctly display your pages that require JavaScript. For example:

```
<script language="JavaScript1.1">
    // Your JavaScript code here
</script>
<noscript>
<hr size="4">
<h1>This Page Requires JavaScript 1.1</h1>
This page requires a browser that supports JavaScript 1.1.<p>
Your browser either does not support JavaScript, or it has JavaScript
support disabled. If you want to correctly view this page, please
upgrade your browser or enable JavaScript support.
```

```
<hr size="4">  
</noscript>
```

There is one problem with the `<noscript>` tag. It was introduced into HTML by Netscape with the release of Netscape 3. Thus, it is not supported in Netscape 2. Since Netscape 2 does not support `<noscript>` and `</noscript>`, it ignores the tags and displays the text that appears between them, even though it does support scripting. In the previous code, however, this works out to our advantage, because we've specified that the code requires JavaScript 1.1 support.

Chapter 21. JavaScript Security

Because of the wide-open nature of the Internet, security is an important issue. This is particularly true with the introduction of languages such as Java and JavaScript, because they allow executable content to be embedded in otherwise static web pages. Since loading a web page can cause arbitrary code to be executed on your computer, stringent security precautions are required to prevent malicious code from doing any damage to your data or your privacy. This chapter discusses Internet security issues related to JavaScript. Note that this chapter does *not* cover any of the many other issues involved in web security, such as the authentication and cryptographic technologies used to keep the contents of web documents and HTML forms private while they traverse the Web.

21.1 JavaScript and Security

JavaScript's first line of defense against malicious code is that the language simply does not support certain capabilities. For example, client-side JavaScript does not provide any way to write or delete files or directories on the client computer. With no File object and no file access functions, a JavaScript program cannot delete a user's data or plant viruses on the user's system.

Similarly, client-side JavaScript has no networking primitives of any type. A JavaScript program can load URLs and can send HTML form data to web servers, CGI scripts, and email addresses, but it cannot establish a direct connection to any other hosts on the network. This means, for example, that a JavaScript program cannot use a client's machine as an attack platform from which to attempt to crack passwords on another machine. (This would be a particularly dangerous possibility if the JavaScript program had been loaded from the Internet through a firewall and could then attempt to break into the intranet protected by the firewall.)

Although the core JavaScript language and the basic client-side object model lack the filesystem and networking features that most malicious code requires, the situation is not quite as simple as it appears. In many web browsers, JavaScript is used as a "script engine" for other software components, such as ActiveX controls in Internet Explorer and plugins in Netscape. These components may have filesystem and network capabilities, and the fact that JavaScript programs can control them clouds the picture and raises security concerns. This is particularly true with ActiveX controls, and Microsoft has at times had to release security patches to prevent JavaScript code from exploiting the capabilities of scriptable ActiveX objects. We'll touch on this issue again briefly at the end of this chapter.

While this intentional lack of features in client-side JavaScript provides a basic level of security against the most egregious attacks, other security issues remain. These are primarily privacy issues -- JavaScript programs must not be allowed to export information about the user of a browser when that information is supposed to be private.

When you browse the Web, one of the pieces of information you are by default consenting to release about yourself is which web browser you use. As a standard part of the HTTP protocol, a string identifying your browser, its version, and its vendor is sent with every request for a web page. This information is public, as is the IP address of your Internet connection, for example. Other information, however, should not be public: this includes your email address, which should not be released unless you choose to do so by sending an email message or authorizing an automated email message to be sent under your name.

Similarly, your browsing history (the record of which sites you've already visited) and the contents of your bookmarks list should remain private. Your browsing history and bookmarks say a lot about your interests; this is information that direct marketers and others pay good money for so that they can target sales pitches to you more effectively. You can be sure that if a web browser or JavaScript allowed this valuable private information to be stolen, some people would steal it every time you visited their sites, and it would be on the market only seconds later. Most web users would be uncomfortable knowing that any site they visited could find out that they were cat fanciers, for example, who were also interested in women's footwear and the Sierra Club.

Even assuming that we have no embarrassing fetishes to hide, there are plenty of good reasons to be concerned about data privacy. One such reason is a pragmatic concern about receiving electronic junk mail (spam) and the like. Another is a legitimate concern about keeping secrets. We don't want a JavaScript program loaded from the Internet and running in one web browser window to be able to start examining the contents of other browser windows that contain pages loaded from the company intranet behind the firewall. The remainder of this chapter explains how JavaScript defends itself against such abuses.

21.2 Restricted Features

As I've already mentioned, the first line of defense against malicious scripts in client-side JavaScript is that the language simply omits certain capabilities. The second line of defense is that JavaScript imposes restrictions on certain features that it does support. For example, client-side JavaScript supports a `close()` method for the Window object, but most (hopefully all) web-browser implementations restrict this method so that a script can close only a window that was opened by a script from the same web server. In particular, a script cannot close a window that the user opened; if it tries to do so, the user is presented with a confirmation box asking if he really wants to close the window.

The most important of these security restrictions is known as the *same-origin policy* and is described in the next section. The following is a list of the other security restrictions found in most implementations of client-side JavaScript. This is not a definitive list. Each browser may have a slightly different set of restrictions, and the proprietary features of each browser may well have proprietary security restrictions to go along with them.

- The History object was originally designed as an array of URLs that represented the complete browsing history of the browser. Once the privacy implications of this became apparent, however, all access to the actual URLs was restricted, and the History object was left with only its `back()`, `forward()`, and `go()` methods to move the browser through the history array without revealing the contents of the array.
- The `value` property of the FileUpload object cannot be set. If this property could be set, a script could set it to any desired filename and cause the form to upload the contents of any specified file (such as a password file) to the server.
- A script cannot submit a form (using the `submit()` method of the Form object, for example) to a `mailto:` or `news:` URL without the user's explicit approval through a confirmation dialog box. Such a form submission would contain the user's email address, which should not be made public without obtaining the user's permission.
- A JavaScript program cannot close a browser window without user confirmation unless it opened the window itself. This prevents malicious scripts from calling `self.close()` to close the user's browsing window, thereby causing the program to exit.
- A script cannot open a window that is smaller than 100 pixels on a side or cause a window to be resized to smaller than 100 pixels on a side. Similarly, such a script cannot move a window off the screen, or create a window that is larger than the screen. This prevents scripts from opening windows that the user cannot see or could easily overlook; such windows could contain scripts that keep running after the user thinks they have stopped. Also, a script may not create a browser window without a titlebar, because such a window could be made to spoof an operating-system dialog box and trick the user into entering a sensitive password, for example.
- A script may not cause a window or frame to display an `about:` URL, such as `about:cache`, because these URLs can expose system information, such as the contents of the browser's cache.
- A script cannot set any of the properties of an Event object. This prevents scripts from spoofing events. A script cannot register event listeners within for or capture events for documents loaded from different sources than the script. This prevents scripts from snooping on the user's input (such as the keystrokes that constitute a password entry) to other pages.

21.3 The Same-Origin Policy

There is one far-reaching security restriction in JavaScript that deserves its own section. This restriction is known as the same-origin policy: a script can read only the properties of windows and documents that have the same origin (i.e., that were loaded from the same host, through the same port, and by the same protocol) as the script itself.

The same-origin policy does not actually apply to all properties of all objects in a window from a different origin. But it does apply to many of them, and in particular, it applies to practically all of the properties of the Document object. For all intents and purposes, you

should consider all predefined properties of all client-side objects with different origins off-limits to your scripts. User-defined properties of objects with different origins may also be restricted, although this may vary from implementation to implementation.

The same-origin policy is a fairly severe restriction, but it is necessary to prevent scripts from stealing proprietary information. Without this restriction, an untrusted script (perhaps a script loaded through a firewall into a browser on a secure corporate intranet) in one window could use DOM methods to read the contents of documents in other browser windows, which might contain private information.

Still, there are circumstances in which the same-origin policy is too restrictive. It poses particular problems for large web sites that use more than one server. For example, a script from *home.netscape.com* might legitimately want to read properties of a document loaded from *developer.netscape.com*, or scripts from *orders.acme.com* might need to read properties from documents on *catalog.acme.com*. To support large web sites of this sort, JavaScript 1.1 introduced the `domain` property of the Document object. By default, the `domain` property contains the hostname of the server from which the document was loaded. You can set this property, but only to a string that is a valid domain suffix of itself. Thus, if `domain` is originally the string "home.netscape.com", you can set it to the string "netscape.com", but not to "home.netscape" or "cape.com", and certainly not to "microsoft.com". (The `domain` value must have at least one dot in it; you cannot set it to "com" or any other top-level domain.)

If two windows (or frames) contain scripts that set `domain` to the same value, the same-origin policy is relaxed for these two windows and each of the windows may read properties from the other. For example, cooperating scripts in documents loaded from *orders.acme.com* and *catalog.acme.com* might set their `document.domain` properties to "acme.com", thereby making the documents appear to have the same origin and enabling each document to read properties of the other.

21.4 Security Zones and Signed Scripts

A one-size-fits-all security policy is never entirely satisfactory. If the policy is too restrictive, trusted scripts don't have the ability to do the interesting and useful things we would like them to do. On the other hand, if the policy is too permissive, untrusted scripts may cause havoc! The ideal solution is to allow the security policy to be configured so that trusted scripts are subject to fewer security restrictions than untrusted scripts. The two major browser vendors, Microsoft and Netscape, have taken different approaches to allowing configurable security; their approaches are briefly described in this section.

Internet Explorer defines "security zones" in which you can list web sites whose scripts you trust and web sites whose scripts you do not trust. You can then configure the security policies of these two zones separately, giving more privileges to and placing fewer restrictions on the trusted sites. (You may also separately configure the privileges of internet and intranet sites that are not explicitly listed in either of the other two zones.)

Unfortunately, this is not a complete or fine-grained solution for JavaScript security, because most of the security options that IE allows you to configure are not directly related to JavaScript. In IE 6 beta, for example, you can specify whether scripts are allowed to control ActiveX objects and Java applets, and whether they can perform paste (as in cut-and-paste) operations. You are not given the option, for example, of disabling the same-origin policy for a trusted site or of allowing scripts from trusted sites to send email messages without a user confirmation.

Netscape 4 and Netscape 6 implement configurable security with an approach known as "signed scripts." Signed scripts provide complete fine-grained configurability of security policies and do it in a way that is cryptographically secure and theoretically very compelling. Unfortunately, since Microsoft has no compatible technology, the process of creating signed scripts is cumbersome for script authors, and the use of signed scripts can be confusing for end users, the use of this promising technology has never really caught on.

Briefly, a signed script bears an unforgeable digital signature that specifies the person or organization that wrote or otherwise takes responsibility for the script. When a signed script needs to circumvent one of the security restrictions described earlier, it first requests a special "privilege" that allows it to do so. When a script requests a privilege, the browser defers to the user. The user is told who the signer of the script is and is asked whether she wants to grant the requested privilege to a script written by that person or organization. Once the user makes the decision, she can have the browser remember it so that she doesn't get asked the same question in the future. In effect, this procedure allows a user to configure a fine-grained customized security policy on the fly, as the need arises.

As I've already mentioned, the process of creating signed scripts is somewhat cumbersome. Also, the details of how it is done have change between Netscape 4 and Netscape 6. Those details are beyond the scope of this book, but you can learn more online at <http://developer.netscape.com/docs/manuals/signedobj/trust/index.htm> and <http://www.mozilla.org/projects/security/components/>.

Chapter 22. Using Java with JavaScript

As we discussed in [Chapter 14](#), Netscape 3 and later and Internet Explorer 4 and later both allow JavaScript programs to read and write the public fields and invoke the public methods of Java applets embedded in HTML documents. Netscape supports JavaScript interaction with Java applets through a technology known as LiveConnect. Internet Explorer instead treats every Java object (including applets) as an ActiveX control and uses its ActiveX scripting technology to allow JavaScript programs to interact with Java. Because Netscape's technology is specifically designed for communication between JavaScript and Java, it has some features that IE's ActiveX technology cannot provide. In practice, however, the two technologies are fairly compatible. Although this chapter is based on Netscape's LiveConnect, the key features it describes work in IE as well.^[1]

^[1] Note that Netscape 6 was released with poor support for LiveConnect but that it is fully implemented in Netscape 6.1 and later.

This chapter begins with a discussion of how you can use JavaScript to script Java applets, how your Java applets can invoke JavaScript code, and how (in Netscape only) you can use JavaScript to work directly with Java system classes. It then documents the nitty-gritty details of how LiveConnect works. It assumes you have at least a basic familiarity with Java programming (see *Java in a Nutshell*, by David Flanagan, and *Learning Java*, by Patrick Niemeyer and Jonathan Knudsen, both published by O'Reilly).

22.1 Scripting Java Applets

As discussed in [Chapter 14](#), all Java applets embedded in a web page become part of the `Document.applets[]` array. Also, if given a `name` or `id`, an applet can be accessed directly as a property of the Document object. For example, the applet created by an `<applet>` tag with a `name` attribute of "chart" can be referred to as `document.chart`.

The public fields and methods of every applet are accessible to JavaScript as if they were the properties and methods of a JavaScript object. For example, if an applet named "chart" defines a field named `lineColor` whose type is `String`, a JavaScript program can query and set this field with code like this:

```
var chartcolor = document.chart.lineColor; // Read an applet field
document.chart.lineColor = "#ff00ff";     // Set an applet field
```

JavaScript can even query and set the values of fields that are arrays. Suppose that the chart applet defines two fields declared as follows (Java code):

```
public int numPoints;
public double[] points;
```

A JavaScript program might use these fields with code like this:

```
for(var i = 0; i < document.chart.numPoints; i++)
    document.chart.points[i] = i*i;
```

This example illustrates the tricky thing about connecting JavaScript and Java: type conversion. Java is a strongly typed language with a fair number of distinct primitive types. JavaScript is loosely typed and has only a single numeric type. In the previous example, a Java integer is converted to a JavaScript number and various JavaScript numbers are converted to Java `double` values. There is a lot of work going on behind the scenes to ensure that these values are properly converted as needed. Later in this chapter, we'll consider the topic of data type conversion in detail.

In addition to querying and setting the fields of a Java applet, JavaScript can also invoke the methods of an applet. Suppose, for example, that the chart applet defines a method named `redraw()`. This method takes no arguments and simply serves to notify the applet that its `points[]` array has been modified and it should redraw itself. JavaScript can invoke this method just as if it was a JavaScript method:

```
document.chart.redraw( );
```

JavaScript can also call methods that take arguments and return values. The underlying LiveConnect or ActiveX scripting technology does the work of converting JavaScript argument values into legal Java values and converting Java return values into legal JavaScript values. Suppose the chart applet defines Java methods like these:

```
public void setDomain(double xmin, double xmax);
public void setChartTitle(String title);
public String getXAxisLabel( );
```

JavaScript can call these methods with code like this:

```
document.chart.setDomain(0, 20);
document.chart.setChartTitle("y = x*x");
var label = document.chart.getXAxisLabel( );
```

Finally, note that Java methods can return Java objects as their return values, and JavaScript can read and write the public fields and invoke the public methods of these objects as well. JavaScript can also use Java objects as arguments to Java methods. Suppose the Java applet defines a method named `getXAxis()` that returns a Java object that is an instance of a class named `Axis` and a method named `setYAxis()` that takes an argument of the same type. Now, suppose further that `Axis` has a method named `setTitle()`. We might use these methods with JavaScript code like this:


```
var xaxis = document.chart.getXAxis( ); // Get an Axis object
var newyaxis = xaxis.clone( ); // Make a copy of it
newyaxis.setTitle("Y"); // Call a method of it...
document.chart.setYAxis(newyaxis); // and pass it to another
method
```

There is one complication when we use JavaScript to invoke the methods of a Java object. Java allows two or more methods to have the same name, as long as they have different argument types. For example, a Java object could declare these two methods:

```
public String convert(int i); // Convert an integer to a string
public String convert(double d); // Convert a floating-point number
```

JavaScript has only one numeric type and doesn't distinguish between integers and floating-point values, so when you use JavaScript to pass a number to the method named "convert", it cannot tell which one you intended to call. In practice, this problem doesn't arise often, and it is usually possible to work around it by simply renaming the methods as needed. The latest versions of LiveConnect (in Netscape 6.1 and later) also allow you to disambiguate cases like this by including the argument types in the method name. For example, if the two methods above were defined by `document.applets[0]`, you could disambiguate them like this:

```
var iconvert = document.applets[0]["convert(int)"]; // Get int method
iconvert(3); // Invoke the method like this
```

22.2 Using JavaScript from Java

Having explored how to control Java from JavaScript code, we now turn to the opposite problem: how to control JavaScript from Java code. This control is accomplished primarily through the Java *netscape.javascript.JSObject* class, which represents a JavaScript object within a Java program. The JavaScript-to-Java capabilities described in the previous section typically work well in both Netscape and Internet Explorer. In contrast, the Java-to-JavaScript techniques described here are not as robustly supported, and you may well encounter bugs in both Netscape and IE.

22.2.1 The JSObject Class

All Java interactions with JavaScript are handled through an instance of the *netscape.javascript.JSObject* class. An instance of this class is a wrapper around a single JavaScript object. The class defines methods that allow you to read and write property values and array elements of the JavaScript object and to invoke methods of the object. Here is a synopsis of this class:

```
public final class JSObject extends Object {
```



```

        // Static method to obtain initial JSObject for applet's browser
window
        public static JSObject getWindow(java.applet.Applet applet);
        public Object getMember(String name);           // Read
object property
        public Object getSlot(int index);             // Read
array element
        public void setMember(String name, Object value); // Set
object property
        public void setSlot(int index, Object value); // Set
array element
        public void removeMember(String name);       // Delete
property
        public Object call(String methodName, Object args[]); // Invoke
method
        public Object eval(String s);                //
Evaluate string
        public String toString( );                  // Convert
to string
        protected void finalize( );
}

```

Because all JavaScript objects appear in a hierarchy rooted in the current browser window, JSObject objects must also appear in a hierarchy. To interact with any JavaScript objects, a Java applet must first obtain a JSObject that represents the browser window (or frame) in which the applet appears. The JSObject class does not define a constructor method, so we cannot simply create an appropriate JSObject. Instead, we must call the static `getWindow()` method. When passed a reference to an applet, this method returns a JSObject that represents the browser window that contains the applet. Thus, every applet that interacts with JavaScript includes a line that looks something like this:

```

JSObject jsroot = JSObject.getWindow(this); // "this" is the applet
itself

```

Having obtained a JSObject that refers to the root window of the JavaScript object hierarchy, you can use instance methods of the JSObject to read the values of properties of the JavaScript object that it represents. Most of these properties have values that are themselves JavaScript objects, so you can continue the process and read their properties as well. The JSObject `getMember()` method returns the value of a named property, while the `getSlot()` method returns the value of a numbered array element of the specified JavaScript object. You might use these methods as follows:

```

import netscape.javascript.JSObject; // This must be at the top of the
file
...
JSObject jsroot = JSObject.getWindow(this); // self
JSObject document = (JSObject) jsroot.getMember("document"); //
.document

```

```

JSObject applets = (JSObject) document.getMember("applets"); //
.applets
Applet applet0 = (Applet) applets.getSlot(0); //
[0]

```

You should note two things about this code fragment. First, `getMember()` and `getSlot()` both return a value of type "Object", which generally must be cast to some more specific value, such as a JSObject. Second, the value read from slot 0 of the `applets` array can be cast to an Applet, rather than a JSObject. This is because the elements of the JavaScript `applets[]` array are JavaScript objects that represent Java Applet objects. When Java reads a JavaScript JavaScript object, it unwraps that object and returns the Java object that it contains (in this case, an Applet). The data conversion that occurs through the JSObject interface is documented later in this chapter.

The JSObject class also supports methods for setting properties and array elements of JavaScript objects. `setMember()` and `setSlot()` are analogous to the `getMember()` and `getSlot()` methods. These methods set the value of a named property or a numbered array element to a specified value. Note, however, that the value to be set must be a Java object. If you want to set a value of a primitive type, use the corresponding Java wrapper class: use an Integer object instead of an `int` value, for example. Finally, the `removeMember()` method allows you to delete the value of a named property from a JavaScript object.

In addition to reading and writing properties and array elements from JavaScript objects, the JSObject class allows you to invoke methods of JavaScript objects. The JSObject `call()` method invokes a named method of the specified JavaScript object and passes a specified array of Java objects as arguments to that method. As we saw when setting JavaScript properties, it is not possible to pass primitive Java values as arguments to a JavaScript method; instead you must use the corresponding Java object types. For example, you might use the `call()` method in Java code like the following to open a new browser window:

```

public JSObject newwin(String url, String window_name)
{
    Object[] args = { url, window_name };
    JSObject win = JSObject.getWindow(this);
    return (JSObject) win.call("open", args);
}

```

The JSObject class has one more important method: `eval()`. This Java method works just like the JavaScript function of the same name -- it executes a string that contains JavaScript code. You'll find that using `eval()` is often much easier than using the various other methods of the JSObject class. Since all the code is passed as a string, you can use string representations of the data types you want -- you do not have to convert Java primitive types to their corresponding object types. For example, compare the following two lines of code that set properties of the main browser window:

```
jsroot.setMember("i", new Integer(0));
jsroot.eval("self.i = 0");
```

The second line is obviously easier to understand. As another example, consider the following use of `eval()` to write a particular frame being displayed in the browser window:

```
JSObject jsroot = JSObject.getWindow(this);
jsroot.eval("parent.frames[1].document.write('Hello from Java!')");
```

To do the equivalent without the `eval()` method is a lot harder:

```
JSObject jsroot = JSObject.getWindow(this);
JSObject parent = (JSObject) jsroot.getMember("parent");
JSObject frames = (JSObject) parent.getMember("frames");
JSObject frame1 = (JSObject) frames.getSlot(1);
JSObject document = (JSObject) frame1.getMember("document");
Object[] args = { "Hello from Java!" };
document.call("write", args);
```

22.2.2 Using JSObjects in Applets

[Example 22-1](#) shows the `init()` method of an applet that uses LiveConnect to interact with JavaScript.

Example 22-1. Using JavaScript from an applet method

```
import netscape.javascript.*

public void init( )
{
    // Get the JSObject representing the applet's browser window.
    JSObject win = JSObject.getWindow(this);

    // Run JavaScript with eval( ). Careful with those nested quotes!
    win.eval("alert('The CPUHog applet is now running on your computer.
" +
            "You may find that your system slows down a bit.');"");
}
```

In order to use any applet, you must compile it and then embed it in an HTML file. When the applet interacts with JavaScript, special instructions are required for both of these steps.

22.2.2.1 Compiling applets that use the JSObject class

Any applet that interacts with JavaScript uses the *netscape.javascript.JSObject* class. To compile such an applet, therefore, your Java compiler must know where to find a definition of this class. Because the class is defined and shipped by Netscape and not by Sun, the *javac* compiler from Sun does not know about it. This section explains how to enable your compiler to find this required class. If you are not using the JDK from Sun, you may have to do something a little different -- see the documentation from the vendor of your Java compiler or Java development environment.

To tell the JDK compiler where to find classes, you set the `CLASSPATH` environment variable. This environment variable specifies a list of directories and JAR files (or ZIP files) that the compiler should search for class definitions (in addition to its standard directory of system classes). The trick is to figure out which JAR file on your system holds the definition of the *netscape.javascript.JSObject* class. In Netscape 6.1, the file is *plugins/java2/javaplugin.jar*, under the Netscape installation directory. In Netscape 4, the file is *java/classes/java40.jar*, under the installation directory. For Netscape 4 on a Windows system, for example, you would probably find *java40.jar* at *C:\Program Files\Netscape\Communicator\Program\Java\Classes\java40.jar*.

For Internet Explorer, the class definition you need is usually in one of the ZIP files in *c:\Windows\Java\Packages*. The trouble is that this directory contains a bunch of ZIP files, all of whose names are gibberish and change from release to release! The largest of the files is typically the one you need. You can use an unzip utility to verify that it contains the file *netscape/javascript/JSObject.class*.

Once you have found the JAR or ZIP file you need, you can tell the compiler about it by setting the `CLASSPATH` environment variable. For a Unix system, set a path like this:

```
setenv CLASSPATH ./usr/local/netscape/plugins/java2/javaplugin.jar
```

And for a Windows system, set a path like this:

```
set CLASSPATH=.;C:\Windows\Java\Packages\5fpnnz7t.zip
```

With `CLASSPATH` set, you should be able to compile your applet with *javac* as you would normally.

22.2.2.2 The *mayscript* attribute

There is an additional requirement for running an applet that interacts with JavaScript. As a security precaution, an applet is not allowed to use JavaScript unless the web page author (who may not be the applet author) explicitly gives the applet permission to do so. To give this permission, you must include the new `mayscript` attribute in the applet's `<applet>` tag in the HTML file.

[Example 22-1](#) showed a fragment of an applet that used JavaScript to display an alert dialog box. Once you have successfully compiled this applet, you might include it in an HTML file as follows:

```
<applet code="CPUHog.class" width="300" height="300"
mayscript></applet>
```

If you do not remember to include the `mayscript` attribute, the applet is not allowed to use the `JSObject` class.

22.3 Using Java Classes Directly

As described in the previous two sections, both Netscape and Internet Explorer allow JavaScript code to interact with Java applets and Java applets to interact with JavaScript. Netscape's LiveConnect technology also allows JavaScript programs to instantiate their own Java objects and use them, even in the absence of any applets. Internet Explorer does not have any analogous capability.

In Netscape, the `Packages` object provides access to all the Java packages that Netscape knows about. The expression `Packages.java.lang` refers to the `java.lang` package, and the expression `Packages.java.lang.System` refers to the `java.lang.System` class. For convenience, `java` is a shortcut for `Packages.java`. In Netscape, JavaScript code might invoke a static method of this `java.lang.System` class as follows:

```
// Invoke the static Java method System.getProperty( )
var javaVersion = java.lang.System.getProperty("java.version");
```

This use of LiveConnect is not limited to system classes, because LiveConnect allows us to use the JavaScript `new` operator to create new instances of Java classes (just as we would in Java). [Example 22-2](#) shows JavaScript code that uses standard Java classes (the JavaScript code looks almost identical to Java code, in fact) to pop up a window and display some text. The result is shown in [Figure 22-1](#).

Figure 22-1. A Java window created from JavaScript



Example 22-2. Scripting the built-in Java classes

```
var f = new java.awt.Frame("Hello World");
var ta = new java.awt.TextArea("hello, world", 5, 20);
```

```
f.add("Center", ta);  
f.pack( );  
f.show( );
```

The code in [Example 22-2](#) creates a simple Java user interface. What is missing, however, is any form of event handling or user interaction. A program like the one shown here is restricted to doing output, since it doesn't include any way for JavaScript to be notified when the user interacts with the Java window. It is possible, though complicated, to use JavaScript to define a Java user interface that responds to events. In Java 1.1 and later, notification of an event is performed by invoking a method of an `EventListener` object. Since Java applets can execute arbitrary strings of JavaScript code, it is possible to define a Java class that implements the appropriate `EventListener` interface and invokes a specified string of JavaScript code when it is notified that an event has occurred. If you create an applet with a method that allows you to create such `EventListener` objects, you can use JavaScript to piece together Java GUIs that include event handlers defined in JavaScript.

Note that LiveConnect does not give complete and unrestricted access to the Java system; in other words, there are some things we cannot do with LiveConnect. For example, LiveConnect does not give us the capability to define new Java classes or subclasses from within JavaScript, nor does it give us the ability to create Java arrays.^[2] In addition to these limitations, access to the standard Java classes is restricted for security reasons. An untrusted JavaScript program cannot use the `java.io.File` class, for example, because that would give it the power to read, write, and delete files on the host system. Untrusted JavaScript code can use Java only in the ways that untrusted applets can.

^[2] JavaScript programs can create arrays indirectly, using the Java 1.1 method `java.lang.reflect.Array.newInstance()`.

22.4 LiveConnect Data Types

To understand how LiveConnect does its job of connecting JavaScript to Java, you have to understand the JavaScript data types that LiveConnect uses. The following sections explain these JavaScript data types. Although Internet Explorer uses a different technology, an understanding of how LiveConnect works will also help you understand the workings of IE. Some of the LiveConnect data types described here have analogs in IE.

22.4.1 The `JavaPackage` Class

A *package* in Java is collection of related Java classes. The `JavaPackage` class is a JavaScript data type that represents a Java package. The properties of a `JavaPackage` are the classes that the package contains (classes are represented by the `JavaClass` class, which we'll see shortly), as well as any other packages that the package contains. There is a restriction on the `JavaPackage` class: you cannot use a JavaScript `for/in` loop to obtain

a complete list of all packages and classes that a `JavaPackage` contains. This restriction is the result of an underlying restriction in the Java virtual machine.

All `JavaPackage` objects are contained within a parent `JavaPackage`; the Window property named `Packages` is a top-level `JavaPackage` that serves as the root of this package hierarchy. It has properties such as `java`, `sun`, and `netscape`, which are `JavaPackage` objects that represent the various hierarchies of Java classes that are available to the browser. For example, the `JavaPackage Packages.java` contains the `JavaPackage Packages.java.awt`. For convenience, every Window object also has `java`, `sun`, and `netscape` properties that are shortcuts to `Packages.java`, `Packages.sun`, and `Packages.netscape`. Thus, instead of typing `Packages.java.awt`, you can simply type `java.awt`.

To continue with the example, `java.awt` is a `JavaPackage` object that contains `JavaClass` objects such as `java.awt.Button`, which represents the `java.awt.Button` class. But it also contains yet another `JavaPackage` object, `java.awt.image`, which represents the `java.awt.image` package in Java.

As you can see, the property naming scheme for the `JavaPackage` hierarchy mirrors the naming scheme for Java packages. Note, however, that there is one big difference between the `JavaPackage` class and the actual Java packages that it represents. Packages in Java are collections of classes, not collections of other packages. That is, `java.lang` is the name of a Java package, but `java` is not. So the `JavaPackage` object named `java` does not actually represent a package in Java -- it is simply a convenient placeholder in the package hierarchy for other `JavaPackage` objects that do represent real Java packages.

On most systems, Java classes are installed in files in a directory hierarchy that corresponds to their package names. For example, the `java.lang.String` class is stored in the file `java/lang/String.class`. Actually, this file is usually contained in a ZIP file, but the directory hierarchy is still there, encoded within the archive. Therefore, instead of thinking of a `JavaPackage` object as representing a Java package, you may find it clearer to think of it as representing a directory or subdirectory in the directory hierarchy of Java classes.

The `JavaPackage` class has a few shortcomings. There is no way for LiveConnect to tell in advance whether a property of a `JavaPackage` refers to a Java class or to another Java package, so JavaScript assumes that it is a class and tries to load a class. Thus, when you use an expression like `java.awt`, LiveConnect first looks for a class file `java/awt.class`. It may even search for this class over the network, causing the web server to log a "404 File Not Found" error. If LiveConnect does not find a class, it assumes that the property refers to a package, but it has no way to ascertain that the package actually exists and has real classes in it. This causes the second shortcoming: if you misspell a class name, LiveConnect happily treats it as a package name, rather than telling you that the class you are trying to use does not exist.

22.4.2 The `JavaClass` Class

The `JavaClass` class is a JavaScript data type that represents a Java class. A `JavaClass` object does not have any properties of its own -- all of its properties represent (and have the same name as) the public static fields and methods of the represented Java class. These public static fields and methods are sometimes called *class fields* and *class methods*, to indicate that they are associated with a class rather than an object instance. Unlike the `JavaPackage` class, `JavaClass` does allow the use of the `for/in` loop to enumerate its properties. Note that `JavaClass` objects do not have properties representing the instance fields and methods of a Java class -- individual instances of a Java class are represented by the `JavaObject` class, which is documented in the next section.

As we saw earlier, `JavaClass` objects are contained in `JavaPackage` objects. For example, `java.lang` is a `JavaPackage` that contains a `System` property. Thus, `java.lang.System` is a `JavaClass` object, representing the Java class `java.lang.System`. This `JavaClass` object, in turn, has properties such as `out` and `in` that represent static fields of the `java.lang.System` class. You can use JavaScript to refer to any of the standard Java system classes in this same way. The `java.lang.Double` class is named `java.lang.Double` (or `Packages.java.lang.Double`), for example, and the `java.awt.Button` class is `java.awt.Button`.

Another way to obtain a `JavaClass` object in JavaScript is to use the `getClass()` function. Given any `JavaObject` object, you can obtain a `JavaClass` object that represents the class of that Java object by passing the `JavaObject` to `getClass()`.^[3]

^[3] Don't confuse the JavaScript `getClass()` function, which returns a `JavaClass` object, with the Java `getClass()` method, which returns a `java.lang.Class` object.

Once you have a `JavaClass` object, there are several things you can do with it. The `JavaClass` class implements the LiveConnect functionality that allows JavaScript programs to read and write the public static fields of Java classes and invoke the public static methods of Java classes. For example, `java.lang.System` is a `JavaClass`. We can read the value of a static field of `java.lang.System` like this:

```
var java_console = java.lang.System.out;
```

Similarly, we might invoke a static method of `java.lang.System` with a line like this one:

```
var java_version = java.lang.System.getProperty("java.version");
```

Recall that Java is a typed language -- all fields and method arguments have types. If you attempt to set a field or pass an argument of the wrong type, an exception is thrown. (Or, in versions of JavaScript prior to 1.5, a JavaScript error occurs.)

There is one more important feature of the `JavaClass` class. You can use `JavaClass` objects with the JavaScript `new` operator to create new instances of Java classes -- i.e., to create `JavaObject` objects. The syntax for doing so is just as it is in JavaScript (and just as it is in Java):

```
var d = new java.lang.Double(1.23);
```

Finally, having created a `JavaObject` in this way, we can return to the `getClass()` function and show an example of its use:

```
var d = new java.lang.Double(1.23);    // Create a JavaObject
var d_class = getClass(d);             // Obtain the JavaClass of the
JavaObject
if (d_class == java.lang.Double) ...;  // This comparison will be true
```

When working with standard system classes like this, you can typically use the name of the system class directly rather than calling `getClass()`. The `getClass()` function is more useful in obtaining the class of a non-system object, such as an applet instance.

Instead of referring to a `JavaClass` with a cumbersome expression like `java.lang.Double`, you can define a variable that serves as a shortcut:

```
var Double = java.lang.Double;
```

This mimics the Java `import` statement and can improve the efficiency of your program, since `LiveConnect` does not have to look up the `lang` property of `java` and the `Double` property of `java.lang`.

22.4.3 The `JavaObject` Class

The `JavaObject` class is a JavaScript data type that represents a Java object. The `JavaObject` class is, in many ways, analogous to the `JavaClass` class. As with `JavaClass`, a `JavaObject` has no properties of its own -- all of its properties represent (and have the same names as) the public instance fields and public instance methods of the Java object it represents. As with `JavaClass`, you can use a JavaScript `for/in` loop to enumerate all the properties of a `JavaObject` object. The `JavaObject` class implements the `LiveConnect` functionality that allows us to read and write the public instance fields and invoke the public methods of a Java object.

For example, if `d` is a `JavaObject` that represents an instance of the `java.lang.Double` class, we can invoke a method of that Java object with JavaScript code like this:

```
n = d.doubleValue( );
```

Similarly, we saw earlier that the *java.lang.System* class has a static field `out`. This field refers to a Java object of class *java.io.PrintStream*. In JavaScript, we can refer to the corresponding `JavaObject` as:

```
java.lang.System.out
```

and we can invoke a method of this object like this:^[4]

^[4] The output of this line of code doesn't appear in the web browser itself, but rather in the Java Console. In Netscape 6, select **Tasks** → **Tools** → **Java Console** to see this window.

```
java.lang.System.out.println("Hello world!");
```

A `JavaObject` object also allows us to read and write the public instance fields of the Java object it represents. Neither the *java.lang.Double* class nor the *java.io.PrintStream* class used in the preceding examples has any public instance fields, however. But suppose we use JavaScript to create an instance of the *java.awt.Rectangle* class:

```
r = new java.awt.Rectangle( );
```

Then we can read and write its public instance fields with JavaScript code like the following:

```
r.x = r.y = 0;
r.width = 4;
r.height = 5;
var perimeter = 2*r.width + 2*r.height;
```

The beauty of LiveConnect is that it allows a Java object, `r`, to be used just as if it were a JavaScript object. Some caution is required, however: `r` is a `JavaObject` and does not behave identically to regular JavaScript objects. The differences will be detailed later. Also, remember that unlike JavaScript, the fields of Java objects and the arguments of their methods are typed. If you do not specify JavaScript values of the correct types, you cause JavaScript errors or exceptions.

In Netscape 6.1 and later the `JavaObject` class makes methods available by name and by name plus argument type, which is useful when there are two or methods that share the same name but expect different types of arguments. As we saw earlier in this chapter, if a `JavaObject` `o` represents an object that has two methods named "convert", the `convert` property of `o` may refer to either of those methods. In recent versions of LiveConnect, however, `o` also defines properties that include the argument types, and you can specify which version of the method you want by including this type information:

```
var iconvert = o["convert(int)"]; // Get the method we want
```

```
iconvert(3); // Invoke it
```

Because the name of the property includes parentheses, you can't use the regular "." notation to access it and must express it as a string within square brackets. The `JavaClass` type has the same capability for overridden static methods.

22.4.4 The `JavaArray` Class

The final LiveConnect data type for JavaScript is the `JavaArray` class. As you might expect by now, instances of this class represent Java arrays and provide the LiveConnect functionality that allows JavaScript to read the elements of Java arrays. Like JavaScript arrays (and like Java arrays), a `JavaArray` object has a `length` property that specifies the number of elements it contains. The elements of a `JavaArray` object are read with the standard JavaScript `[]` array index operator. They can also be enumerated with a `for/in` loop. You can use `JavaArray` objects to access multidimensional arrays (actually arrays of arrays), just as in JavaScript or Java.

For example, suppose we create an instance of the `java.awt.Polygon` class:

```
p = new java.awt.Polygon( );
```

The `JavaObject` `p` has properties `xpoints` and `ypoints` that are `JavaArray` objects representing Java arrays of integers. (To learn the names and types of these properties, look up the documentation for `java.awt.Polygon` in a Java reference manual.) We can use these `JavaArray` objects to randomly initialize the Java polygon with code like this:

```
for(var i = 0; i < p.xpoints.length; i++)  
    p.xpoints[i] = Math.round(Math.random( ) * 100);  
for(var i = 0; i < p.ypoints.length; i++)  
    p.ypoints[i] = Math.round(Math.random( ) * 100);
```

22.4.5 Java Methods

The `JavaClass` and `JavaObject` classes allow us to invoke static methods and instance methods, respectively. In Netscape 3, Java methods were internally represented by a `JavaMethod` object. In Netscape 4, however, Java methods are simply native methods, like the methods of built-in JavaScript objects such as `String` and `Date`.

When you're using Java methods, remember that they expect a fixed number of arguments of fixed types. If you pass the wrong number of arguments, or an argument of the wrong type, you cause a JavaScript error.

22.5 LiveConnect Data Conversion

Java is a strongly typed language with a relatively large number of data types, while JavaScript is an untyped language with a relatively small number of types. Because of this major structural difference between the two languages, one of the central responsibilities of LiveConnect is data conversion. When JavaScript sets a Java class or instance field or passes an argument to a Java method, a JavaScript value must be converted to an equivalent Java value, and when JavaScript reads a Java class or instance field or obtains the return value of a Java method, that Java value must be converted into a compatible JavaScript value.^[5]

^[5] In addition, data conversion must happen when Java reads or writes a JavaScript field or invokes a JavaScript method. These conversions are done differently, however, and are described later in this chapter, when we discuss how to use JavaScript from Java. For now, we're considering only the data conversion that happens when JavaScript code interacts with Java, not the other way around.

[Figure 22-2](#) and [Figure 22-3](#) illustrate how data conversion is performed when JavaScript writes Java values and when it reads them, respectively.

Figure 22-2. Data conversions performed when JavaScript writes Java values

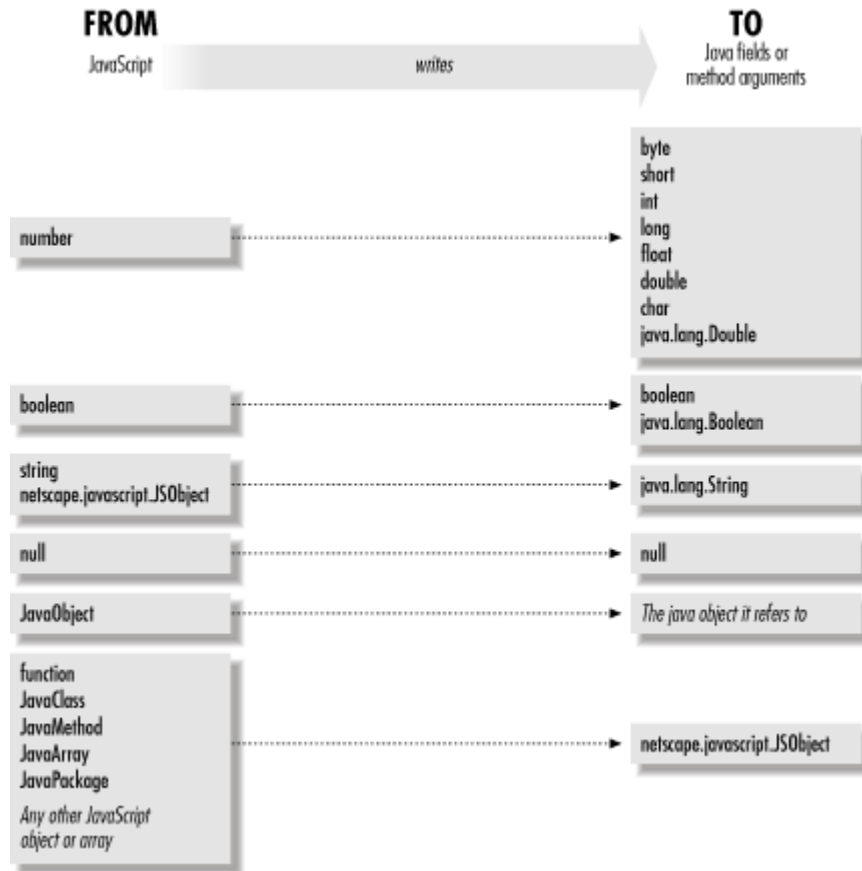
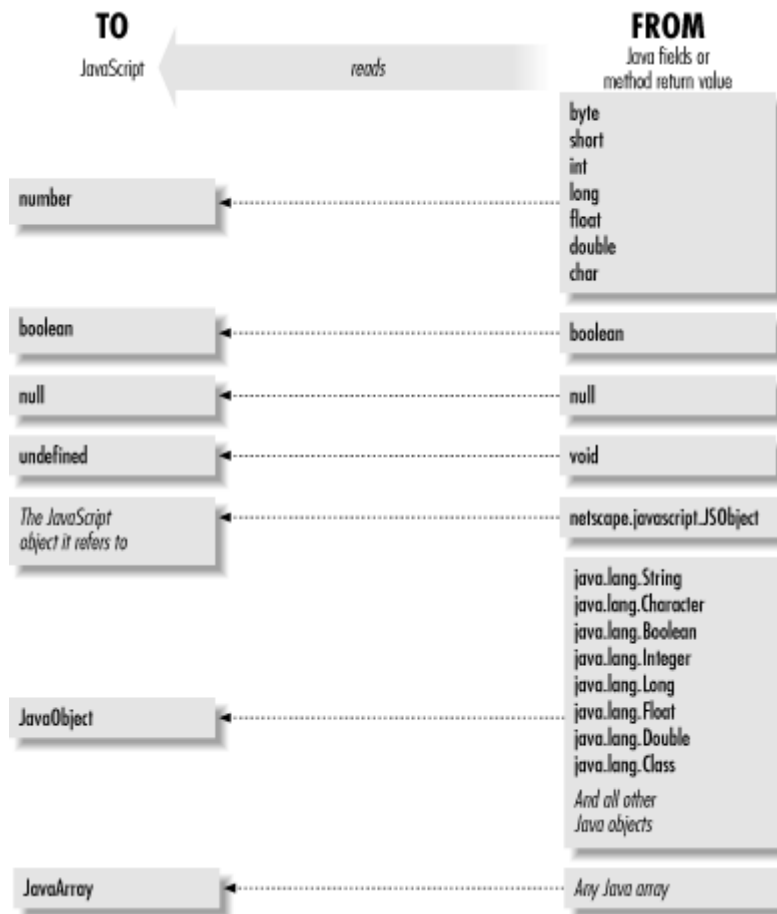


Figure 22-3. Data conversions performed when JavaScript reads Java values



Notice the following points about the data conversions illustrated in [Figure 22-2](#):

- [Figure 22-2](#) does not show all possible conversions between JavaScript types and Java types. This is because JavaScript-to-JavaScript type conversions can occur before the JavaScript-to-Java conversion takes place. For example, if you pass a JavaScript number to a Java method that expects a *java.lang.String* argument, JavaScript first converts that number to a JavaScript string, which can then be converted to a Java string.
- A JavaScript number can be converted to any of the primitive Java numeric types. The actual conversion performed depends, of course, on the type of the Java field being set or the method argument being passed. Note that you can lose precision doing this, for example, when you pass a large number to a Java field of type `short` or when you pass a floating-point value to a Java integral type.
- A JavaScript number can also be converted to an instance of the Java class *java.lang.Double* but not to an instance of a related class, such as *java.lang.Integer* or *java.lang.Float*.
- JavaScript does not have any representation for character data, so a JavaScript number may also be converted to the Java primitive `char` type.

- A `JavaScript` `JavaScript` object is "unwrapped" when passed to Java -- that is, it is converted to the Java object it represents. Note, however, that `JavaScript` objects in `JavaScript` are not converted to instances of `java.lang.Class`, as might be expected.
- `JavaScript` arrays are not converted to Java arrays.

Also notice these points about the conversions illustrated in [Figure 22-3](#):

- Since `JavaScript` does not have a type for character data, the Java primitive `char` type is converted to a `JavaScript` number, not a string, as might be expected.
- A Java instance of `java.lang.Double`, `java.lang.Integer`, or a similar class is not converted to a `JavaScript` number. Like any Java object, it is converted to a `JavaScript` object in `JavaScript`.
- A Java string is an instance of `java.lang.String`, so like any other Java object, it is converted to a `JavaScript` object rather than to an actual `JavaScript` string.
- Any type of Java array is converted to a `JavaScript` object in `JavaScript`.

22.5.1 Wrapper Objects

Another important concept that you must grasp in order to fully understand [Figure 22-2](#) and [Figure 22-3](#) is the idea of wrapper objects. While conversions between most `JavaScript` and Java primitive types are possible, conversions between object types are generally not possible. This is why `LiveConnect` defines the `JavaScript` object in `JavaScript` -- it represents a Java object that cannot be directly converted to a `JavaScript` object. In a sense, a `JavaScript` object is a `JavaScript` wrapper around a Java object. When `JavaScript` reads a Java value (a field or the return value of a method), any Java objects are wrapped and `JavaScript` sees a `JavaScript` object.

A similar thing happens when `JavaScript` writes a `JavaScript` object into a Java field or passes a `JavaScript` object to a Java method. There is no way to convert the `JavaScript` object to a Java object, so the object gets wrapped. The Java wrapper for a `JavaScript` object is the Java class `netscape.javascript.JSObject`.

Things get interesting when these wrapper objects are passed back. If `JavaScript` writes a `JavaScript` object into a Java field or passes it to a Java method, `LiveConnect` first unwraps the object, converting the `JavaScript` object back into the Java object that it represents. Similarly, if `JavaScript` reads a Java field or gets the return value of a Java method that is an instance of `netscape.javascript.JSObject`, that `JSObject` is also unwrapped to reveal and return the original `JavaScript` object.

22.5.2 LiveConnect Data Conversion in Netscape 3

In `Netscape 3`, there was a bug in the way that `LiveConnect` converted Java values to `JavaScript` values: the value of a primitive field of a Java object was incorrectly returned as a `JavaScript` object, rather than as a `JavaScript` primitive value. For example, if `JavaScript` read the value of a field of type `int`, `LiveConnect` in `Netscape 3` converted that

value to a Number object, rather than to a primitive numeric value. Similarly, LiveConnect converted the value of Java `boolean` fields to JavaScript Boolean objects, rather than primitive JavaScript boolean values. Note that this bug occurred only when querying the values of Java fields. It did not occur when LiveConnect converted the return value of a Java method.

Number and Boolean objects in JavaScript behave almost, but not exactly, the same as primitive number and boolean values. One important difference is that Number objects, like all JavaScript objects, use the `+` operator for string concatenation rather than for addition. As a result, code like the following that uses LiveConnect in Netscape 3 can yield unexpected results:

```
var r = new java.awt.Rectangle(0,0,5,5);
var w = r.width;      // This is a Number object, not a primitive number.
var new_w = w + 1;   // Oops! new_w is now "51", not 6, as expected.
```

To work around this problem, you can explicitly call the `valueOf()` method to convert a Number object to its corresponding numeric value. For example:

```
var r = new java.awt.Rectangle(0,0,5,5);
var w = r.width.valueOf(); // Now we've got a primitive number.
var new_w = w + 1;        // This time, new_w is 6, as desired.
```

22.6 JavaScript Conversion of JavaObjects

Having worked your way through the previous dense section, you may hope that we are done with the topic of data conversion. Unfortunately, there is more to be discussed on the topic of how JavaScript converts JavaObject objects to various JavaScript primitive types. Notice in [Figure 22-3](#) that quite a few Java data types, including Java strings (instances of `java.lang.String`), are converted to JavaObject objects in JavaScript rather than being converted to actual JavaScript primitive types, such as strings. This means that when you use LiveConnect, you'll often be working with JavaObject objects.

Refer back to [Table 11-1](#), which shows how various JavaScript data types are converted when used in various contexts. For example, when a number is used in a string context, it is converted to a string, and when an object is used in a boolean context, it is converted to the value `false` if it is `null` and `true` otherwise. These conversion rules don't apply to JavaObject objects, which are converted using their own rules, as follows:

- When a JavaObject is used in a numeric context, it is converted to a number by invoking the `doubleValue()` method of the Java object it represents. If the Java object does not define this method, a JavaScript error occurs.

- When a `JavaScriptObject` is used in a boolean context, it is converted to a boolean value by invoking the `booleanValue()` method of the Java object it represents. If the Java object does not define this method, a JavaScript error occurs.
- When a `JavaScriptObject` is used in a string context, it is converted to a string value by invoking the `toString()` method of the Java object it represents. All Java objects define or inherit this method, so this conversion always succeeds.
- When a `JavaScriptObject` is used in an object context, no conversion is necessary, since it is already a JavaScript object.

Because of these different conversion rules, and for other reasons as well, `JavaScriptObject` objects behave differently than other JavaScript objects, and there are some common pitfalls that you need to recognize. First, it is not uncommon to work with a `JavaScriptObject` that represents an instance of `java.lang.Double` or some other numeric object. In many ways, such a `JavaScriptObject` behaves like a primitive number value, but be careful when using the `+` operator. When you use a `JavaScriptObject` (or any JavaScript object) with `+`, you are specifying a string context, so the object is converted to a string for string concatenation instead of being converted to a number for addition.

When you want to explicitly convert a JavaScript object to a primitive value, you usually call its `valueOf()` method. Note that this does not work with `JavaScriptObject` objects. As we discussed earlier, the `JavaScriptObject` class defines no properties of its own; all of its properties represent fields and methods of the Java object it represents. This means that `JavaScriptObject` objects don't support common JavaScript methods, such as `valueOf()`. In the case of our `JavaScriptObject`-wrapped `java.lang.Double` object, you should call the Java `doubleValue()` method when you need to force the object into a primitive value.

Another difference between `JavaScriptObject` objects and other JavaScript data types is that `JavaScriptObjects` can be used in a boolean context only if they define a `booleanValue()` method. Suppose `button` is a JavaScript variable that may contain `null` or may hold a `JavaScriptObject` that represents an instance of the `java.awt.Button` class. If you want to check whether the variable contains `null`, you might write code like this, out of habit:

```
if (!button) { ... }
```

If `button` is `null`, this works fine. But if `button` actually contains a `JavaScriptObject` representing a `java.awt.Button` instance, LiveConnect tries to invoke the `booleanValue()` method. When it discovers that the `java.awt.Button` class doesn't define one, it causes a JavaScript error. The workaround in this case is to be explicit about what you are testing for, to avoid using the `JavaScriptObject` in a boolean context:

```
if (button != null) { ... }
```

This is a good habit to get into, in any case, since it makes your code easier to read and understand.

22.7 Java-to-JavaScript Data Conversion

In the last two sections, we discussed the rules by which values are converted when JavaScript reads and writes Java fields and invokes Java methods. Those rules explained how the JavaScript `JavaScript`, `JavaScriptArray`, and `JavaScriptClass` objects convert data; they apply only to the case of JavaScript manipulating Java. When Java manipulates JavaScript, the conversion is performed by the Java `JSObject` class, and the conversion rules are different. [Figure 22-4](#) and [Figure 22-5](#) illustrate these conversions.

Figure 22-4. Data conversions performed when Java writes JavaScript values

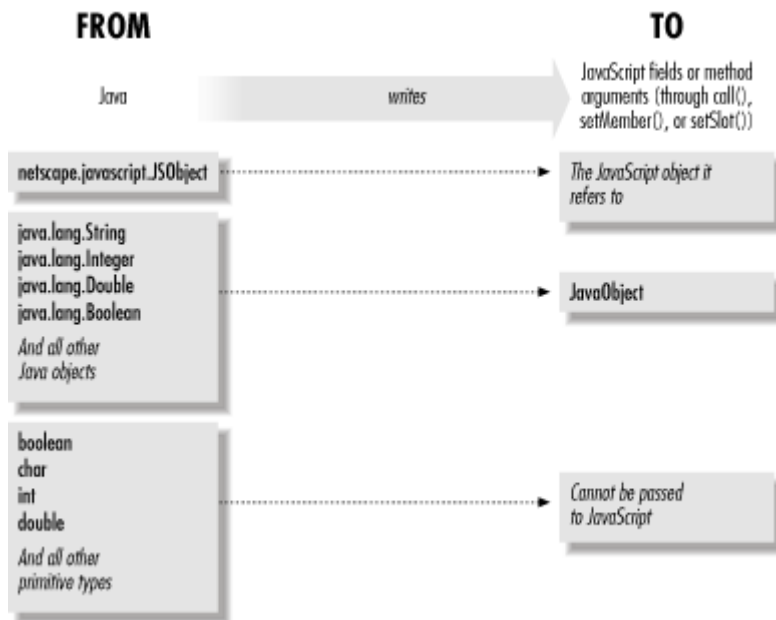
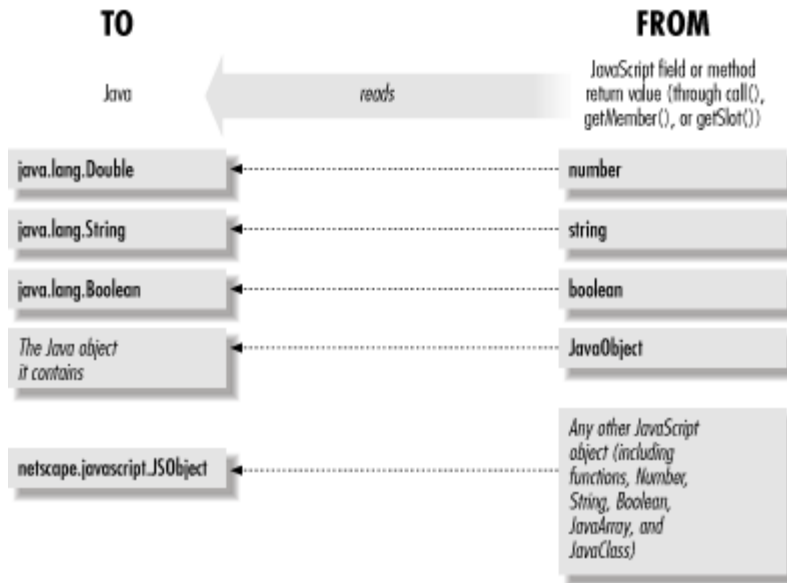


Figure 22-5. Data conversions performed when Java reads JavaScript values



The point to remember when studying these figures is that Java can interact with JavaScript only through the API provided by the `JSObject` class. Because Java is a strongly typed language, the methods defined by this class can work only with Java objects, not with primitive values. For example, when you read the value of a JavaScript number, the `getMember()` method returns a `java.lang.Double` object, rather than a primitive `double` value.

When writing JavaScript functions that are invoked from Java, bear in mind that the arguments passed by Java are either JavaScript objects from unwrapped Java `JSObjects`, or `JavaObjects`. LiveConnect simply does not allow Java to pass primitive values as method arguments. As we saw earlier in this chapter, `JavaObject` objects behave somewhat differently than other objects. For example, an instance of `java.lang.Double` behaves differently than a primitive JavaScript number or even a JavaScript `Number` object. The same caution applies when you are working with JavaScript properties that have their values set by Java.

One way to avoid the whole issue of data conversion is to use the `eval()` method of the `JSObject` class whenever your Java code wants to communicate with JavaScript. In order to do this, your Java code must convert all method arguments or property values to string form. Then, the string to be evaluated can be passed unchanged to JavaScript, which can convert the string form of the data to the appropriate JavaScript data values.

Part III: Core JavaScript Reference

This part of the book is a complete reference to all of the objects, properties, functions, methods, and event handlers in the core JavaScript language. The first few pages of this part explain how to use this reference material.

NOTE:

Since this E-Book was Ripped, Every page of text was high-lighted, copied, and pasted into a word document. Then converted to PDF.

The reference is an extra 450 pages, and TONS of time would be required.... So (since I have a life) I will **NOT** be ripping this section of the book.

Sorry... the book ends here, a very successful 459 pages, enjoy, you can learn everything you need to know about JavaScript...

■ lilmeanman -- www.magic-html.com –
www.suprnova.org –
www.torrentbox.com –

www.malinko.net –
www.warezfreaks.com –
www.emudimension.com



