

Activate Your Web Pages

4th Edition
Covers JavaScript 1.5



JavaScript

The Definitive Guide

O'REILLY®

David Flanagan

Tableau EX1044

IPR2024-01388

U.S. Patent No. 9,712,595

FOURTH EDITION

JavaScript

The Definitive Guide

David Flanagan

O'REILLY*

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

JavaScript: The Definitive Guide, Fourth Edition
by David Flanagan

Copyright © 2002, 1998, 1997, 1996 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Paula Ferguson
Production Editor: Rachel Wheeler
Cover Designer: Edie Freedman
Interior Designer: David Futato

Printing History:

August 1996: Beta Edition.
January 1997: Second Edition.
June 1998: Third Edition.
January 2002: Fourth Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *JavaScript: The Definitive Guide, Fourth Edition*, the image of a Java rhinoceros, and related trade dress are trademarks of O'Reilly Media, Inc. Java™, all Java-based trademarks and logos, and JavaScript™ are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems.

Netscape, Navigator, and the Netscape Communications Corporate Logos are trademarks and tradenames of Netscape Communications Corporation. Internet Explorer and the Internet Explorer Logo are trademarks and tradenames of Microsoft Corporation. All other product names and logos are trademarks of their respective owners.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00048-0

[M]

[8/05]

Table of Contents

Preface	xiii
---------------	------

1. Introduction to JavaScript	1
JavaScript Myths	2
Versions of JavaScript	2
Client-Side JavaScript	3
JavaScript in Other Contexts	5
Client-Side JavaScript: Executable Content in Web Pages	6
Client-Side JavaScript Features	8
JavaScript Security	12
Example: Computing Loan Payments with JavaScript	12
Using the Rest of This Book	16
Exploring JavaScript	18

Part I. Core JavaScript

2. Lexical Structure	23
Character Set	23
Case Sensitivity	24
Whitespace and Line Breaks	24
Optional Semicolons	24
Comments	25
Literals	26
Identifiers	26
Reserved Words	27

3. Data Types and Values	29
Numbers	30
Strings	33
Boolean Values	36
Functions	37
Objects	39
Arrays	40
null	42
undefined	42
The Date Object	43
Regular Expressions	43
Error Objects	44
Primitive Data Type Wrapper Objects	44
4. Variables	46
Variable Typing	46
Variable Declaration	47
Variable Scope	48
Primitive Types and Reference Types	50
Garbage Collection	52
Variables as Properties	53
Variable Scope Revisited	54
5. Expressions and Operators	56
Expressions	56
Operator Overview	57
Arithmetic Operators	60
Equality Operators	62
Relational Operators	65
String Operators	67
Logical Operators	68
Bitwise Operators	70
Assignment Operators	72
Miscellaneous Operators	73
6. Statements	79
Expression Statements	79
Compound Statements	80

if	81
else if	82
switch	83
while	86
do/while	87
for	87
for/in	89
Labels	90
break	90
continue	92
var	93
function	93
return	95
throw	95
try/catch/finally	96
with	98
The Empty Statement	99
Summary of JavaScript Statements	100
7. Functions	102
Defining and Invoking Functions	102
Functions as Data	106
Function Scope: The Call Object	108
Function Arguments: The Arguments Object	109
Function Properties and Methods	111
8. Objects	114
Objects and Properties	114
Constructors	116
Methods	117
Prototypes and Inheritance	120
Object-Oriented JavaScript	123
Objects as Associative Arrays	130
Object Properties and Methods	132
9. Arrays	138
Arrays and Array Elements	138
Array Methods	142

10. Pattern Matching with Regular Expressions	147
Defining Regular Expressions	147
String Methods for Pattern Matching	156
The RegExp Object	158
11. Further Topics in JavaScript	161
Data Type Conversion	161
By Value Versus by Reference	166
Garbage Collection	171
Lexical Scoping and Nested Functions	173
The Function() Constructor and Function Literals	175
Netscape's JavaScript 1.2 Incompatibilities	175

Part II. Client-Side JavaScript

12. JavaScript in Web Browsers	181
The Web Browser Environment	181
Embedding JavaScript in HTML	185
Execution of JavaScript Programs	193
13. Windows and Frames	199
Window Overview	199
Simple Dialog Boxes	201
The Status Line	203
Timeouts and Intervals	204
Error Handling	206
The Navigator Object	207
The Screen Object	209
Window Control Methods	209
The Location Object	213
The History Object	215
Multiple Windows and Frames	218
14. The Document Object	225
Document Overview	225
Dynamically Generated Documents	230
Document Color Properties	235
Document Information Properties	235

Forms	236
Images	236
Links	243
Anchors	245
Applets	247
Embedded Data	248
15. Forms and Form Elements	249
The Form Object	250
Defining Form Elements	251
Scripting Form Elements	255
Form Verification Example	263
16. Scripting Cookies	266
An Overview of Cookies	266
Storing Cookies	268
Reading Cookies	269
Cookie Example	270
17. The Document Object Model	274
An Overview of the DOM	274
Using the Core DOM API	285
DOM Compatibility with Internet Explorer 4	303
DOM Compatibility with Netscape 4	305
Convenience Methods: The Traversal and Range APIs	306
18. Cascading Style Sheets and Dynamic HTML	314
Styles and Style Sheets with CSS	315
Element Positioning with CSS	322
Scripting Styles	332
DHTML in Fourth-Generation Browsers	341
Other DOM APIs for Styles and Style Sheets	345
19. Events and Event Handling	351
Basic Event Handling	352
Advanced Event Handling with DOM Level 2	361
The Internet Explorer Event Model	376
The Netscape 4 Event Model	382

20. Compatibility Techniques	387
Platform and Browser Compatibility	387
Language Version Compatibility	392
Compatibility with Non-JavaScript Browsers	396
21. JavaScript Security	399
JavaScript and Security	399
Restricted Features	400
The Same-Origin Policy	402
Security Zones and Signed Scripts	403
22. Using Java with JavaScript	405
Scripting Java Applets	405
Using JavaScript from Java	407
Using Java Classes Directly	411
LiveConnect Data Types	413
LiveConnect Data Conversion	418
JavaScript Conversion of JavaObjects	421
Java-to-JavaScript Data Conversion	423
Part III. Core JavaScript Reference	
Core JavaScript Reference	427
Part IV. Client-Side JavaScript Reference	
Client-Side JavaScript Reference	541
Part V. W3C DOM Reference	
W3C DOM Reference	685
Part VI. Class, Property, Method, and Event Handler Index	
Class, Property, Method, and Event Handler Index	857
Index	871

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.

Errata

Please help us at O'Reilly & Associates, Inc. improve future editions and future printings of this book by reporting any errors, inaccuracies, bugs, misleading or confusing statements, and plain old typos that you find anywhere in this book. O'Reilly maintains a web site for this book that includes a listing of all known errors. You can find the errata list by following a link from the book's catalog page:

<http://www.oreilly.com/catalog/jsript4/>

The errata page includes a link to a form that allows you to report any bugs you find. You can also report bugs or ask questions about this book by sending email to:

bookquestions@oreilly.com

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/jsript4/>

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

To comment or ask technical questions about this and other O'Reilly books, you can also send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

<http://www.oreilly.com>

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

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."

Table 1-1. *Versions of JavaScript*

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 <code>Array</code> object; most serious bugs resolved. Implemented by Netscape 3.
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. Conformance 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. Conformance 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 Chapters 17, 18, and 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 1.0/2.0	Form manipulation
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 ECMA-Script 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.* 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.

1.5 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.

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.† 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.

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

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

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

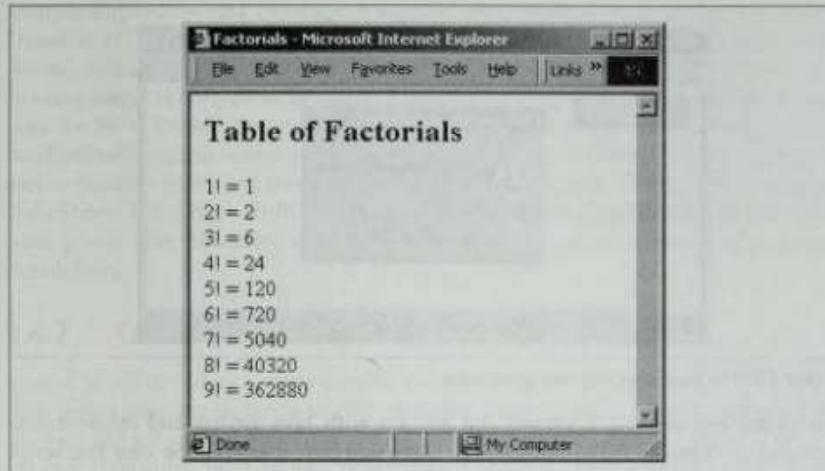


Figure 1-1. A web page generated with JavaScript

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.

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.

Examples 1-1 and 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



Figure 1-2. The JavaScript response to an event

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 `<img>` tag to produce image roll-over 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 stand-alone 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.

JavaScript Loan Calculator - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Enter Loan Information:

1) Amount of the loan (any currency): 200000

2) Annual percentage rate of interest: 6.5

3) Repayment period in years: 30

Compute

Payment Information:

4) Your monthly payment will be: 1264.14

5) Your total payment will be: 455088.98

6) Your total interest payments will be: 255088.98

Done My Computer

Figure 1-3. A JavaScript loan payment calculator

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>

```

Example 1-3. Computing loan payments with JavaScript (continued)

```

        <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. Chapters 2 through 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, *Lexical Structure*, explains the basic structure of the language.
- Chapter 3, *Data Types and Values*, documents the data types supported by JavaScript.
- Chapter 4, *Variables*, covers variables, variable scope, and related topics.
- Chapter 5, *Expressions and Operators*, 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, *Statements*, 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, *Functions*, documents how functions are defined, invoked, and manipulated in JavaScript.
- Chapter 8, *Objects*, 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, *Arrays*, describes the creation and use of arrays in JavaScript.
- Chapter 10, *Pattern Matching with Regular Expressions*, explains how to use regular expressions in JavaScript to perform pattern-matching and search-and-replace operations.
- Chapter 11, *Further Topics in JavaScript*, 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, *JavaScript in Web Browsers*, 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, *Windows and Frames*, documents the most central and important object of client-side JavaScript, the Window object, as well as several important window-related objects.
- Chapter 14, *The Document Object*, explains the Document object and related objects that expose the contents of an HTML document to JavaScript code.
- Chapter 15, *Forms and Form Elements*, 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, *Scripting Cookies*, illustrates the use of cookies to save state in web programming.
- Chapter 17, *The Document Object Model*, 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, *Cascading Style Sheets and Dynamic HTML*, 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, *Events and Event Handling*, 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, *Compatibility Techniques*, 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, *JavaScript Security*, enumerates the security restrictions built into client-side JavaScript and explains the rationale for them.
- Chapter 22, *Using Java with JavaScript*, 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.

Parts III, IV, and 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!

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, “Timeouts and Intervals,” and Section 13.5, “Error Handling,” 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>`, `<br>`, `<font>`, `<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>

Table 19-1. Event handlers and the HTML elements that support them (continued)

Handler	Triggered when	Supported by
onchange	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>
onclick	Mouse press and release; follows mouseup event. Return false 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 elements
ondblclick	Double-click.	HTML4, N6, IE4: most elements
onerror	Error when loading image.	N3, IE4: <code></code>
onfocus	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>
onkeydown	Key pressed down. Return false to cancel.	N4: <code><input></code> , <code><textarea></code> HTML4, N6, IE4: form elements and <code><body></code>
onkeypress	Key pressed and released. Return false to cancel.	N4: <code><input></code> , <code><textarea></code> HTML4, N6, IE4: form elements and <code><body></code>
onkeyup	Key released.	N4: <code><input></code> , <code><textarea></code> HTML4, N6, IE4: form elements and <code><body></code>
onload	Document load complete.	HTML4, N2, IE3: <code><body></code> , <code><frameset></code> N3, IE4: <code></code> N6, IE4: <code><iframe></code> , <code><object></code>
onmousedown	Mouse button pressed.	N4: <code><a></code> , <code><area></code> , <code></code> HTML4, N6, IE4: most elements
onmousemove	Mouse moved.	HTML4, N6, IE4: most elements
onmouseout	Mouse moves off element.	N3: <code><a></code> , <code><area></code> HTML4, N6, IE4: most elements
onmouseover	Mouse moves over element. For links, return true to prevent URL from appearing in status bar.	N2, IE3: <code><a></code> , <code><area></code> HTML4, N6, IE4: most elements
onmouseup	Mouse button released.	N4: <code><a></code> , <code><area></code> , <code></code> HTML4, N6, IE4: most elements
onreset	Form reset requested. Return false to prevent reset.	HTML4, N3, IE4: <code><form></code>
onresize	Window size changes.	N4, IE4: <code><body></code> , <code><frameset></code>
onselect	Text selected.	HTML4, N6, IE3: <code><input></code> , <code><textarea></code>
onsubmit	Form submission requested. Return false to prevent submission.	HTML4, N3, IE4: <code><form></code>
onunload	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.*

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

* 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.

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 0 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 lower-case 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, "Scope of Event Handlers.")

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 0 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 “Event Handlers and the `this` Keyword”). 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, "Lexical Scoping and Nested Functions.")

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 `MouseEvents` 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
MouseEvents	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 `MouseEvents` 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>

Table 19-3. Event types (continued)

Event type	Interface	B	C	Supported by/detail properties
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, 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
DOMNodeInsertedIntoDocument	MutationEvent	no	no	none
DOMNodeRemoved	MutationEvent	yes	no	relatedNode
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 mouse-down, 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.
```

Example 19-2. Dragging with the DOM Level 2 event model (continued)

```

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("MouseEvent");
```

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 HTML-Events 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
```

Example 19-3. Dragging with the IE event model (continued)

```

// 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 capture events by calling
// setCapture() on the element to capture them.
elementToDrag.attachEvent("onmousemove", moveHandler);
elementToDrag.attachEvent("onmouseup", upHandler);
elementToDrag.setCapture();
}
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.
 */

```

Example 19-3. Dragging with the IE event model (continued)

```
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
        elementToDrag.detachEvent("onmouseup", upHandler);
        elementToDrag.detachEvent("onmousemove", moveHandler);
        elementToDrag.releaseCapture();
    }
    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 non-functional.

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.*

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
  }
}
```

* 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 (continued)

```
        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>
```

JavaScript: The Definitive Guide



JavaScript is a powerful, object-based scripting language; JavaScript programs can be embedded directly in HTML web pages. When combined with the Document Object Model (DOM) defined by a web browser, JavaScript allows you to create Dynamic HTML (DHTML) content and interactive client-side web applications. JavaScript syntax is based on the popular programming languages C, C++, and Java, which makes it familiar and easy to learn for experienced programmers. At the same time, JavaScript is an interpreted scripting language, providing a flexible, forgiving environment in which new programmers can learn.

JavaScript: The Definitive Guide provides a thorough description of the core JavaScript language and both the legacy and standard DOMs implemented in web browsers. The book includes sophisticated examples that show you how to handle common tasks, such as validating form data, working with cookies, and creating portable DHTML animations. The book also contains detailed reference sections that cover the core JavaScript API, the legacy client-side API, and the W3C standard DOM API, documenting every JavaScript object, method, property, constructor, constant, function, and event handler in those APIs.

This fourth edition of the bestselling JavaScript book has been carefully updated to cover JavaScript 1.5 (ECMAScript Version 3). The book also provides complete coverage of the W3C DOM standard (Level 1 and Level 2), while retaining material on the legacy Level 0 DOM for backward compatibility.

JavaScript: The Definitive Guide is a complete programmer's guide and reference manual for JavaScript. It is particularly useful for developers working with the latest standards-compliant web browsers, such as Internet Explorer 6, Netscape 6, and Mozilla. HTML authors can learn how to use JavaScript to build dynamic web pages. Experienced programmers can quickly find the information they need to start writing sophisticated JavaScript programs. This book is an indispensable reference for all JavaScript programmers, regardless of experience level.

*"A must-have reference for expert JavaScript programmers...
Well-organized and detailed."*

—Brendan Eich
Creator of JavaScript



Visit O'Reilly on the Web at www.oreilly.com

